US007796128B2

(12) United States Patent
Radzikowski et al.

(10) Patent No.: US 7,796,128 B2
(45) Date of Patent: Sep. 14, 2010

(54) DYNAMICALLY LOAD BALANCING GAME PHYSICS USING REAL-TIME OBJECT SCALING

(75) Inventors: **Eric Michael Radzikowski**, Franklin, WI (US); **Robert Allen Shearer**, Rochester, MN (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 667 days.

(21) Appl. No.: **11/674,734**

(22) Filed: **Feb. 14, 2007**

(65) **Prior Publication Data**

US 2008/0192047 A1 Aug. 14, 2008

(51) **Int. Cl.**
G06T 15/00 (2006.01)

(52) **U.S. Cl.** **345/419**; 345/420; 345/421; 345/426; 345/427; 345/473; 345/949; 345/950; 345/951; 345/958

(58) **Field of Classification Search** 345/419, 345/420, 421, 426, 427, 473, 949, 950, 951, 345/958; 428/34
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,305,430	A *	4/1994	Glassner	345/427
5,336,537	A *	8/1994	Bovone	428/34
5,949,430	A *	9/1999	Robertson et al.	345/619
5,986,675	A *	11/1999	Anderson et al.	345/473
6,091,422	A *	7/2000	Ouaknine et al.	345/419
6,373,485	B2 *	4/2002	Sowizral et al.	345/421
6,791,549	B2 *	9/2004	Hubrecht et al.	345/473
2007/0182732	A1 *	8/2007	Woop et al.	345/420
2009/0167763	A1 *	7/2009	Waechter et al.	345/426

OTHER PUBLICATIONS

Laszlo szecsi et al. "Accelerating Animation Through Verification of Shooting Walks", ACM 2003, pp. 231-237.*
Wald et al. "Distributed Interactive Ray Tracing of Dynamic scenes", IEEE 2003, pp. 77-85.*

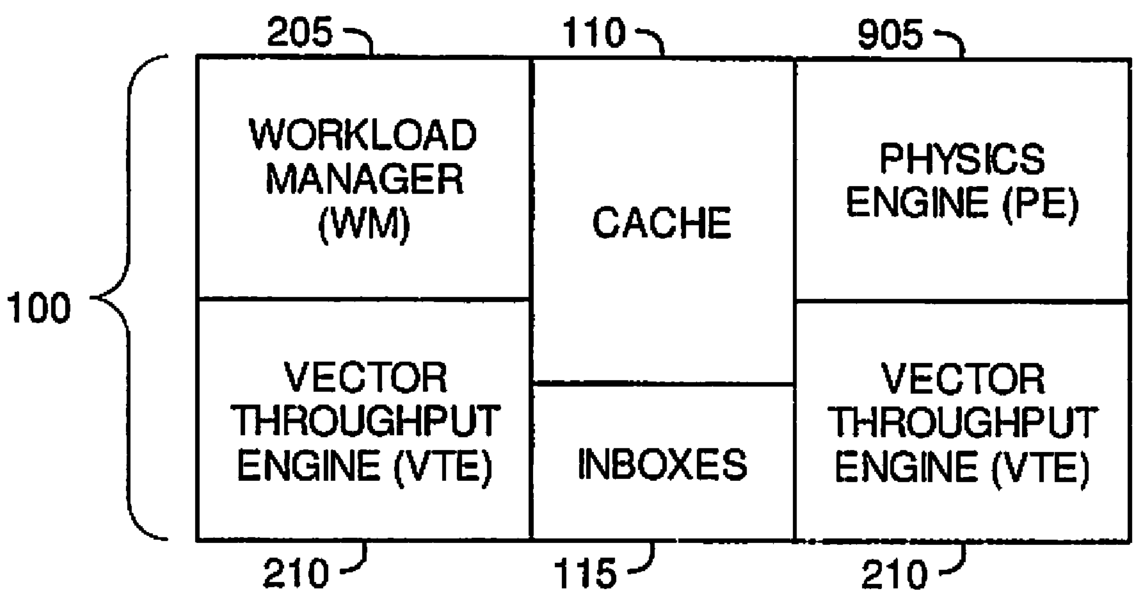
* cited by examiner

Primary Examiner—Kimbinh T Nguyen
(74) *Attorney, Agent, or Firm*—Patterson & Sheridan LLP

(57) **ABSTRACT**

The present invention provides methods and apparatus for dynamically varying a physics workload by scaling the classification of objects within a three-dimensional scene. According to embodiments of the invention, a physics engine may classify a plurality of objects within a three-dimensional scene as either static objects or as dynamic objects. The physics engine may then perform physics operations with the objects, and may monitor the workload experienced by processing elements within a system which includes the physics engine. Based on the workload experienced by the processing elements within the system, the physics engine may vary the number of objects which are classified as dynamic and vary the number of objects which are classified as static. By varying the classification of the objects, the physics engine may vary the workload experienced by the physics engine.

20 Claims, 20 Drawing Sheets



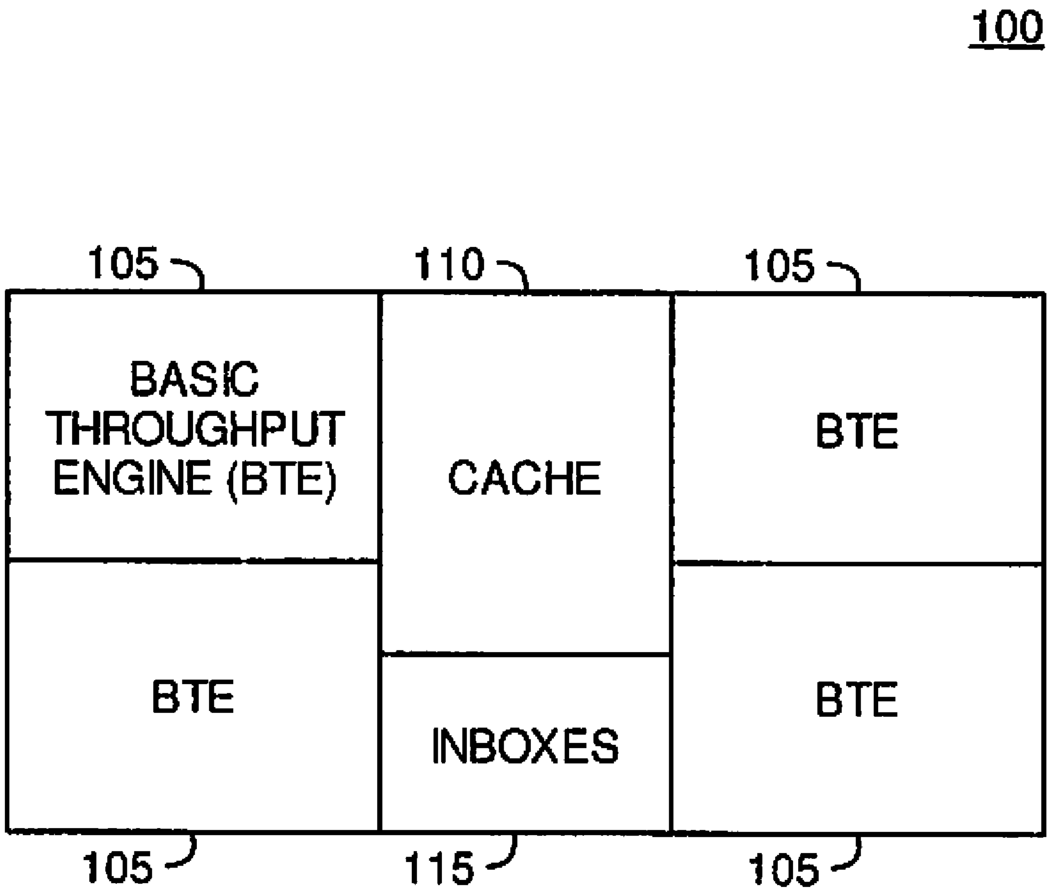

100
105
110
105
BASIC THROUGHPUT ENGINE (BTE)
CACHE
BTE
BTE
INBOXES
BTE
105
115
105

FIG. 1

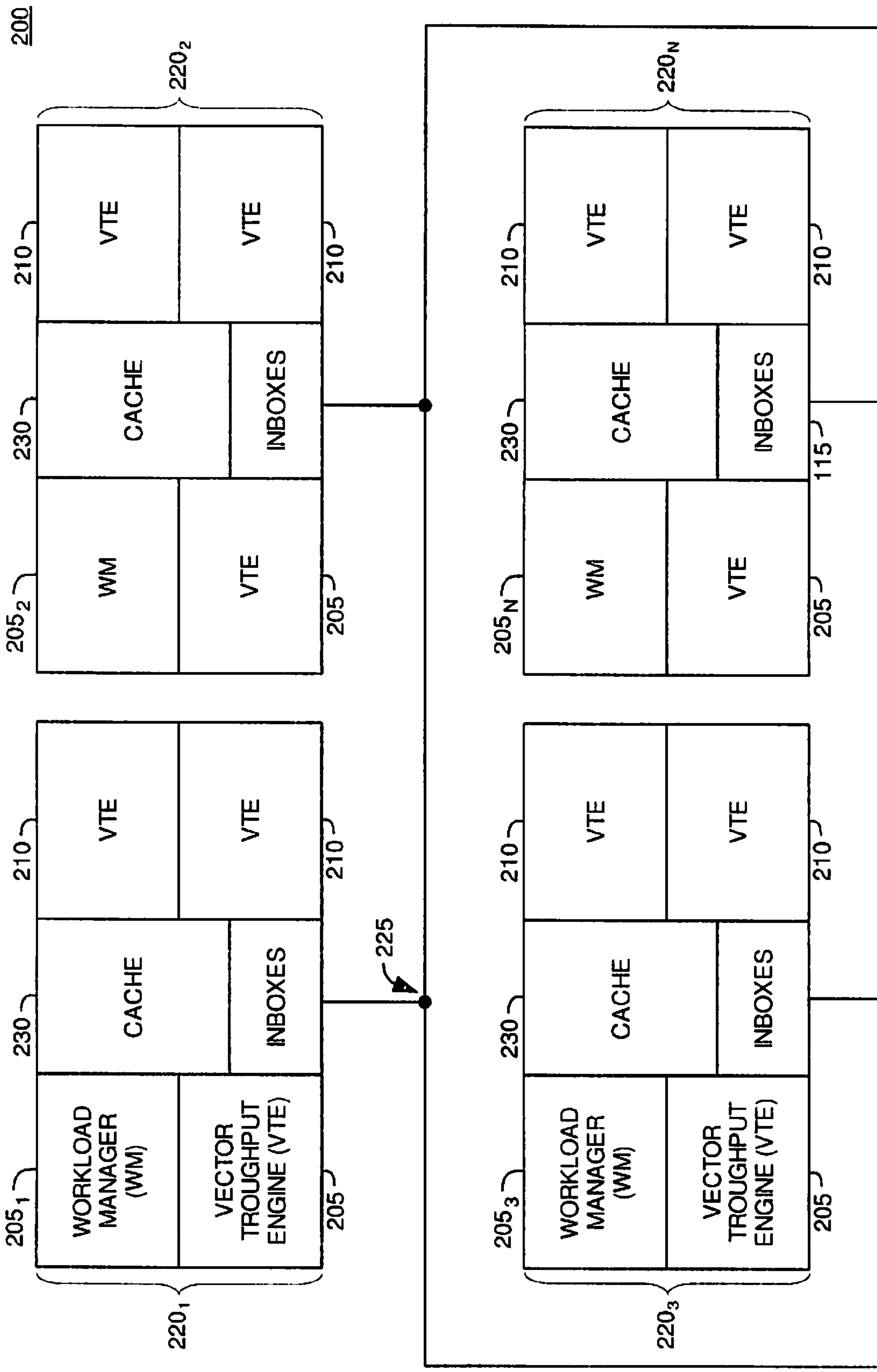

200
220_1
220_2
220_3
220_N
205_1
205_2
205_3
205_N
205
210
230
225
115
WORKLOAD MANAGER (WM)
VECTOR TROUGHPUT ENGINE (VTE)
CACHE
INBOXES
VTE
WM

FIG. 2

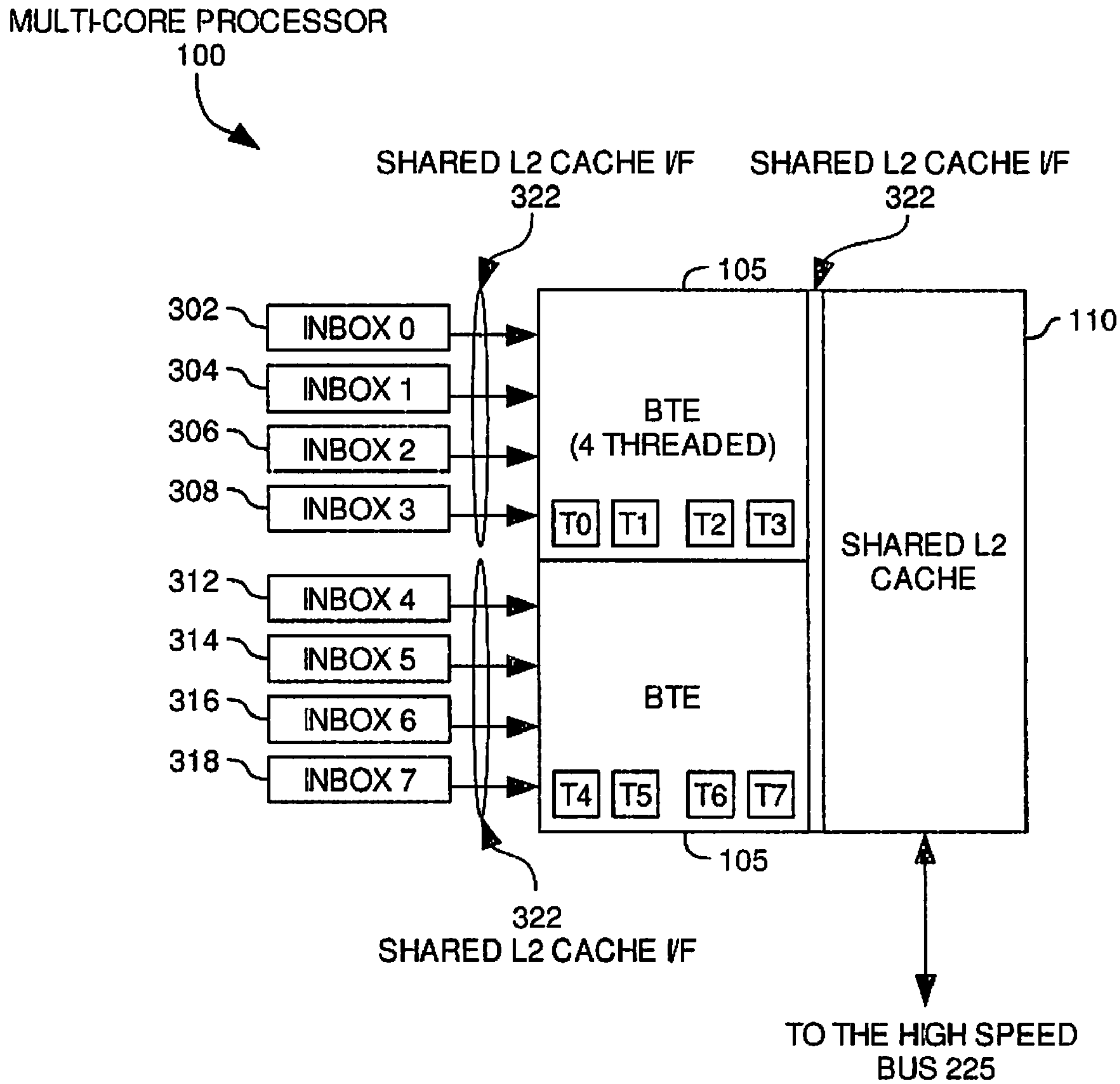

MULTI-CORE PROCESSOR
100
SHARED L2 CACHE I/F
322
SHARED L2 CACHE I/F
322
105
110
302
INBOX 0
304
INBOX 1
306
INBOX 2
308
INBOX 3
BTE
(4 THREADED)
T0
T1
T2
T3
SHARED L2
CACHE
312
INBOX 4
314
INBOX 5
316
INBOX 6
318
INBOX 7
BTE
T4
T5
T6
T7
105
322
SHARED L2 CACHE I/F
TO THE HIGH SPEED
BUS 225

FIG. 3A

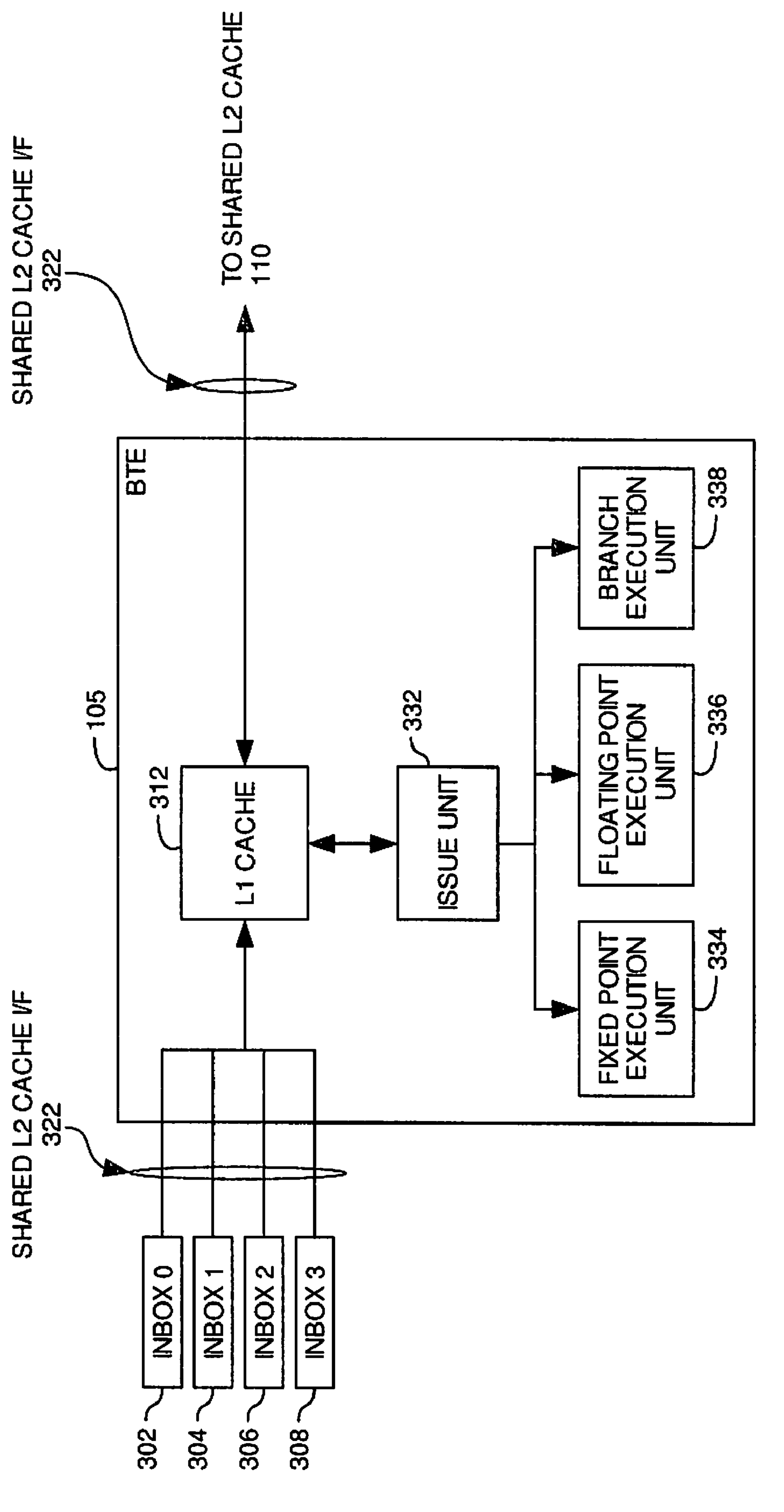

SHARED L2 CACHE I/F
322
TO SHARED L2 CACHE
110
BTE
105
312
L1 CACHE
332
ISSUE UNIT
BRANCH EXECUTION UNIT
338
FLOATING POINT EXECUTION UNIT
336
FIXED POINT EXECUTION UNIT
334
SHARED L2 CACHE I/F
322
INBOX 0
INBOX 1
INBOX 2
INBOX 3
302
304
306
308

FIG. 3B

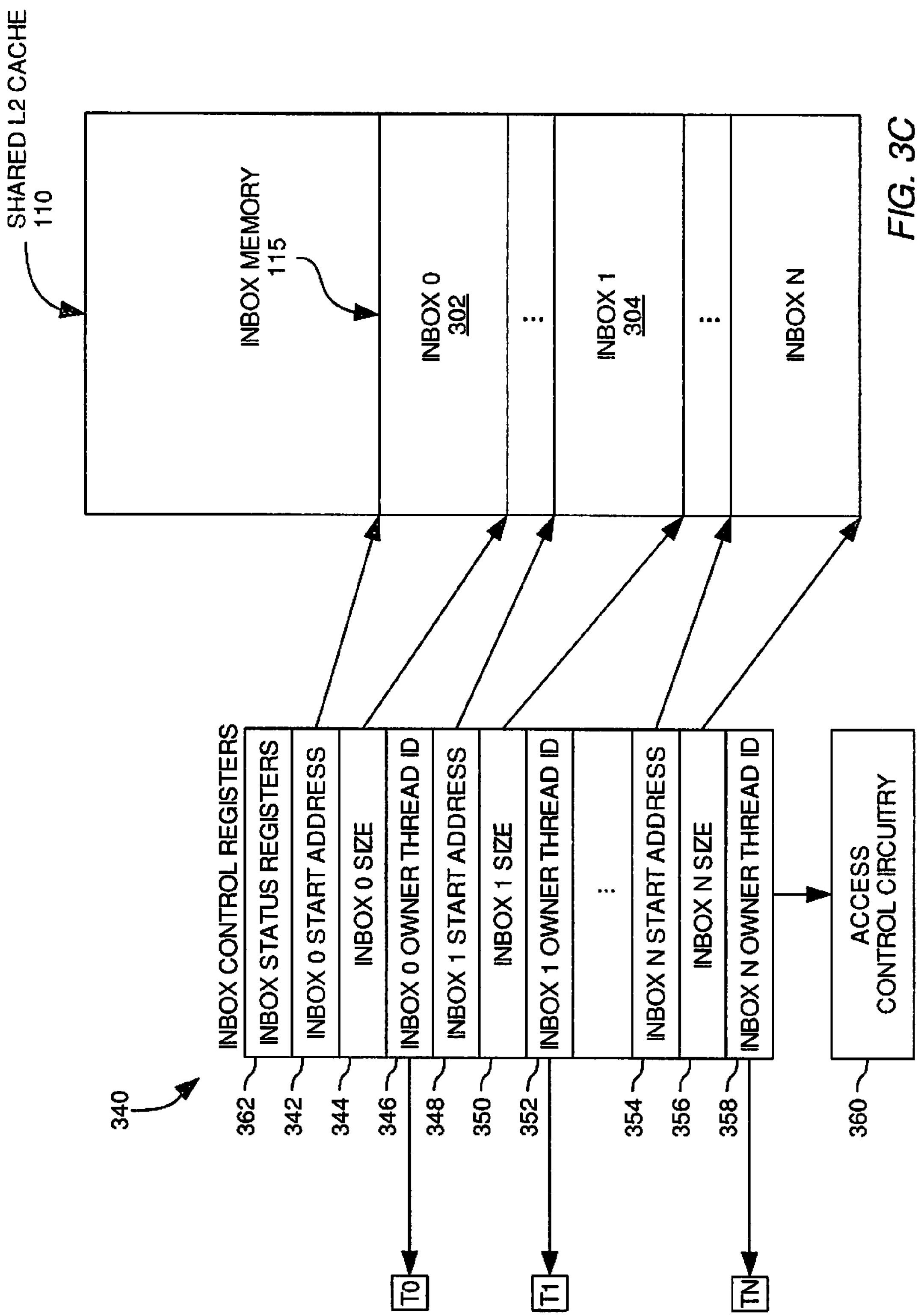

SHARED L2 CACHE
110
INBOX MEMORY
115
INBOX 0
302
...
INBOX 1
304
...
INBOX N
340
INBOX CONTROL REGISTERS
362
INBOX STATUS REGISTERS
342
INBOX 0 START ADDRESS
344
INBOX 0 SIZE
346
INBOX 0 OWNER THREAD ID
348
INBOX 1 START ADDRESS
350
INBOX 1 SIZE
352
INBOX 1 OWNER THREAD ID
...
354
INBOX N START ADDRESS
356
INBOX N SIZE
358
INBOX N OWNER THREAD ID
360
ACCESS
CONTROL CIRCUITRY
T0
T1
TN

FIG. 3C

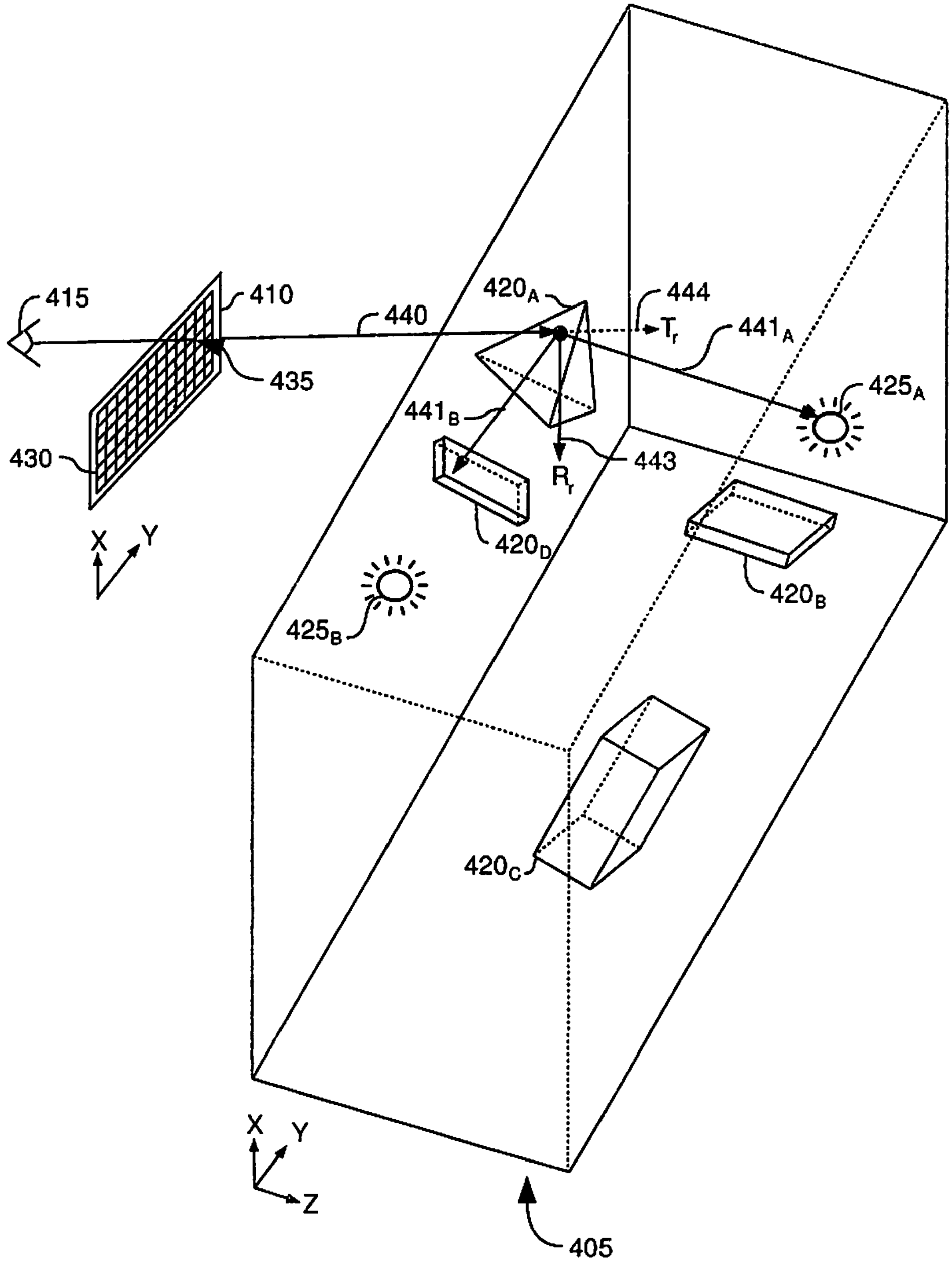
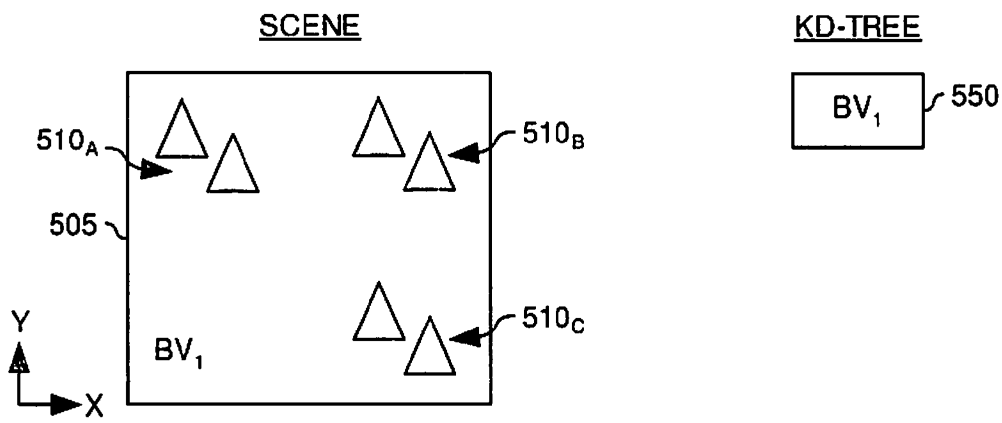
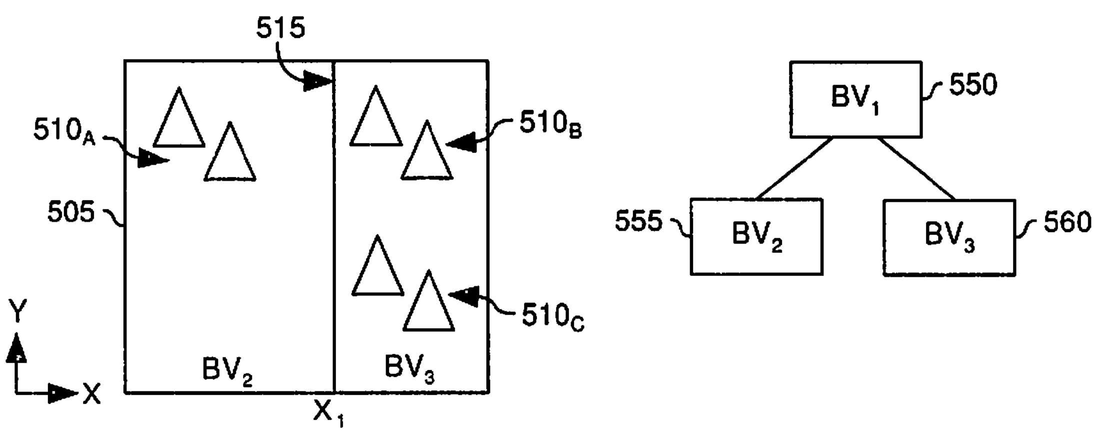

415
410
440
420_A
444
T_r
441_A
435
425_A
441_B
430
443
R_r
420_D
X
Y
420_B
425_B
420_C
X
Y
Z
405

FIG. 4



SCENE
KD-TREE
BV_1
550
510_A
510_B
505
510_C
Y
X
BV_1

FIG. 5A



515
510_A
510_B
505
510_C
Y
X
BV_2
BV_3
X_1
BV_1
550
555
BV_2
BV_3
560

FIG. 5B

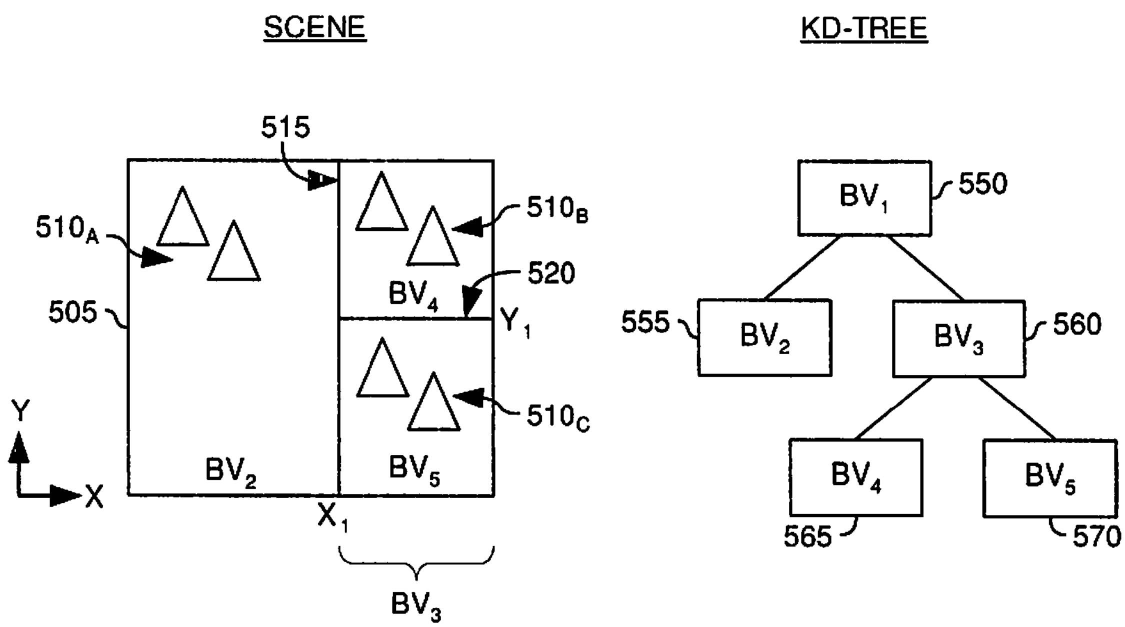

SCENE
KD-TREE
515
510_A
510_B
520
505
BV_4
Y_1
510_C
Y
X
BV_2
BV_5
X_1
BV_3
BV_1
550
555
BV_2
BV_3
560
BV_4
BV_5
565
570

FIG. 5C

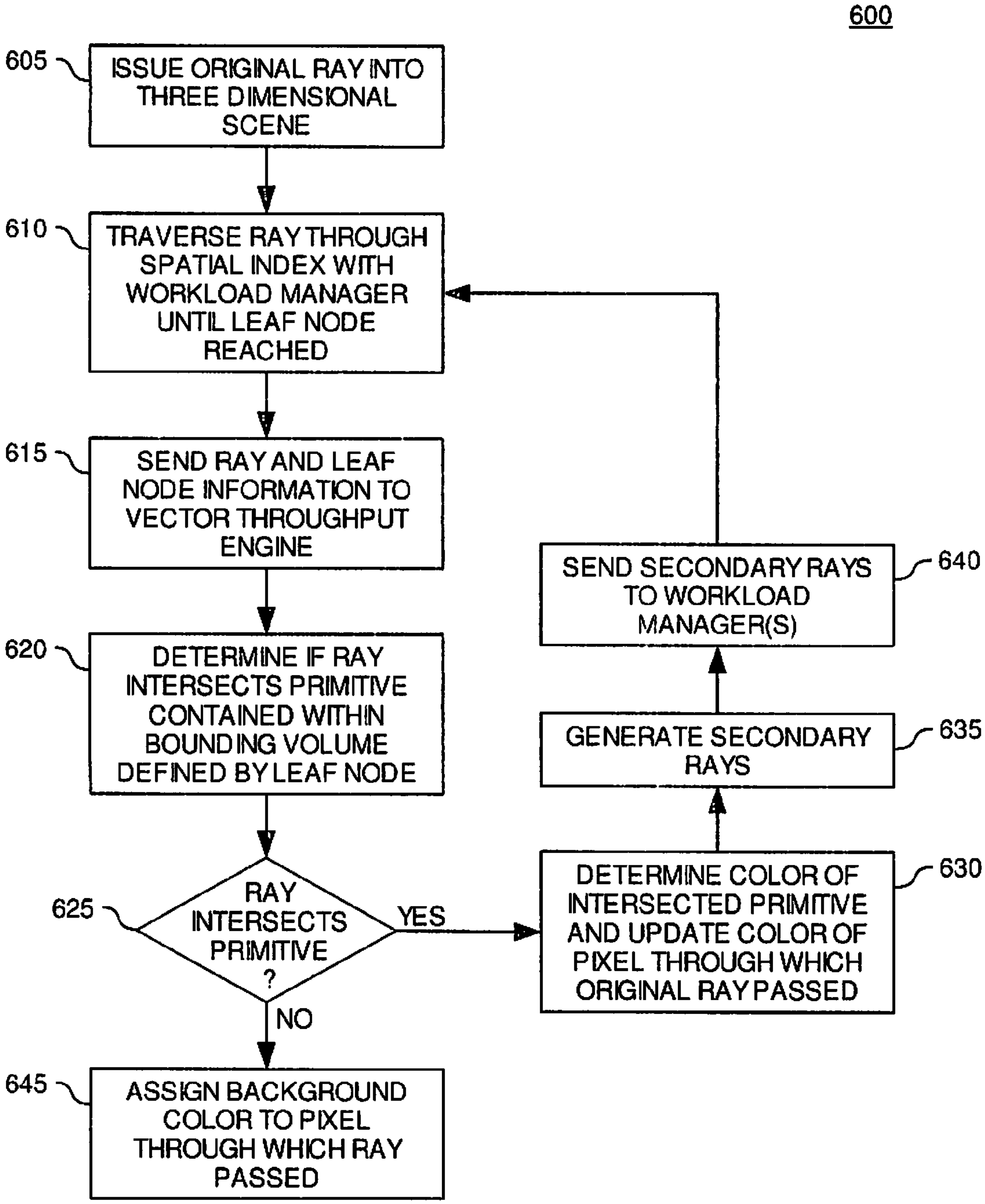

600
605
ISSUE ORIGINAL RAY INTO THREE DIMENSIONAL SCENE
610
TRAVERSE RAY THROUGH SPATIAL INDEX WITH WORKLOAD MANAGER UNTIL LEAF NODE REACHED
615
SEND RAY AND LEAF NODE INFORMATION TO VECTOR THROUGHPUT ENGINE
620
DETERMINE IF RAY INTERSECTS PRIMITIVE CONTAINED WITHIN BOUNDING VOLUME DEFINED BY LEAF NODE
625
RAY INTERSECTS PRIMITIVE ?
YES
NO
630
DETERMINE COLOR OF INTERSECTED PRIMITIVE AND UPDATE COLOR OF PIXEL THROUGH WHICH ORIGINAL RAY PASSED
635
GENERATE SECONDARY RAYS
640
SEND SECONDARY RAYS TO WORKLOAD MANAGER(S)
645
ASSIGN BACKGROUND COLOR TO PIXEL THROUGH WHICH RAY PASSED

FIG. 6

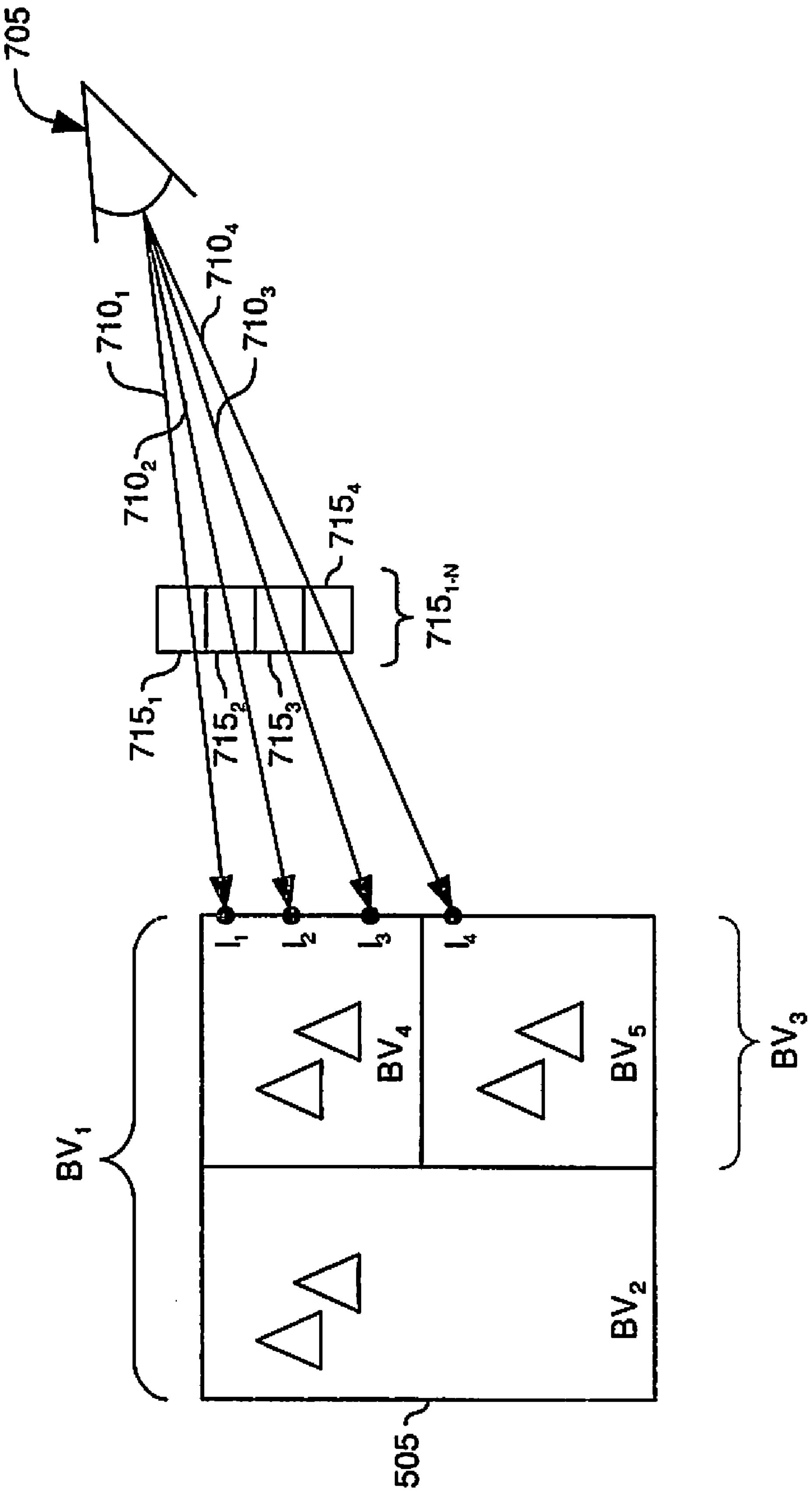
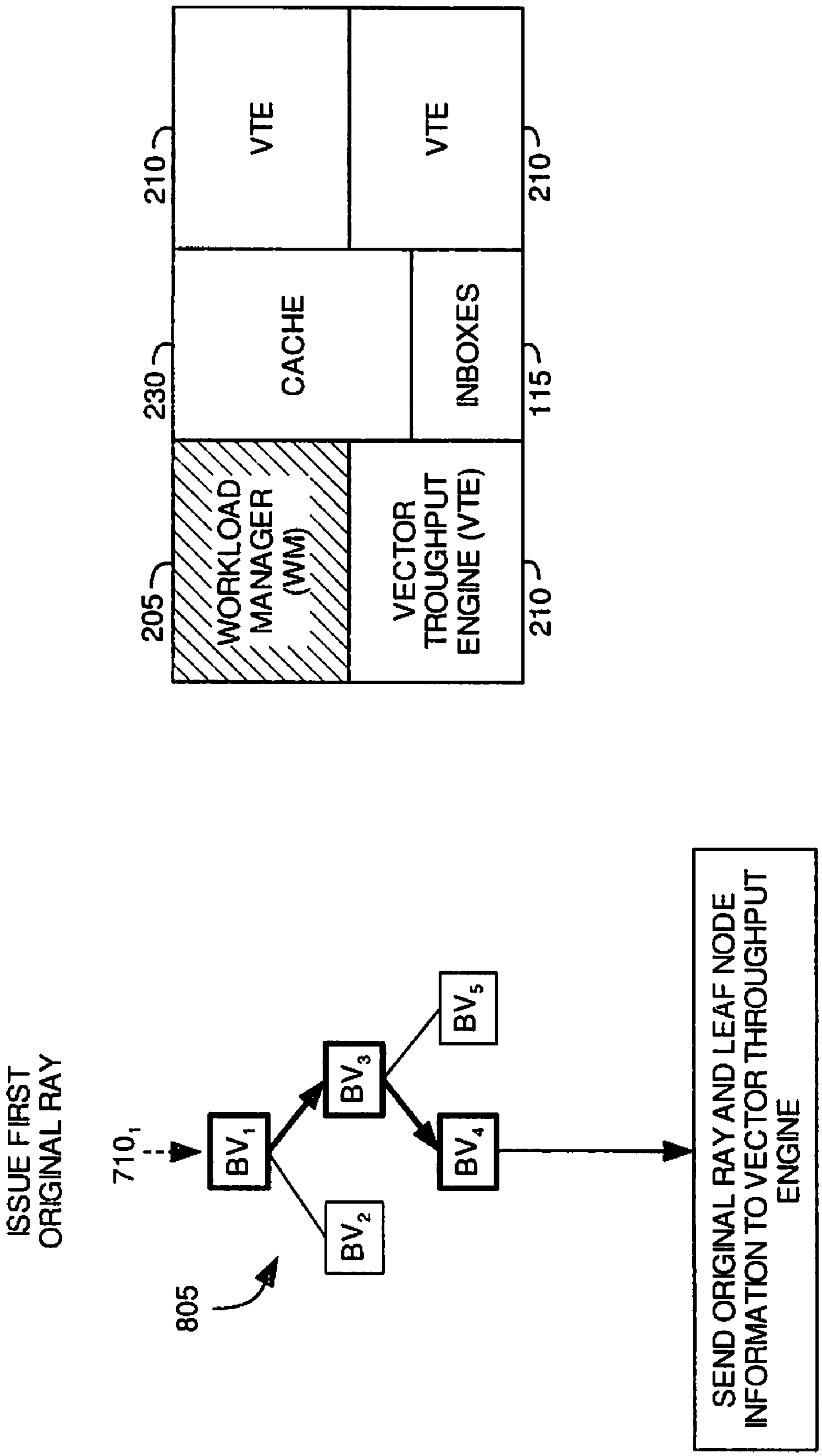

705
710_1
710_2
710_3
710_4
715_4
715_{1-N}
715_1
715_2
715_3
I_1
I_2
I_3
I_4
BV_1
BV_2
BV_3
BV_4
BV_5
505

FIG. 7



WORKLOAD MANAGER (WM)
205
VECTOR TROUGHPUT ENGINE (VTE)
210
CACHE
230
INBOXES
115
VTE
210
VTE
210
ISSUE FIRST ORIGINAL RAY
710_1
805
BV_1
BV_2
BV_3
BV_4
BV_5
SEND ORIGINAL RAY AND LEAF NODE INFORMATION TO VECTOR THROUGHPUT ENGINE

FIG. 8A

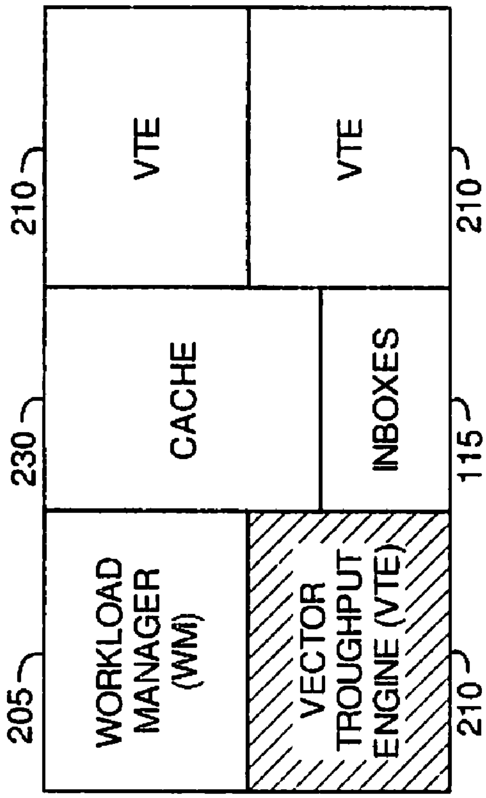
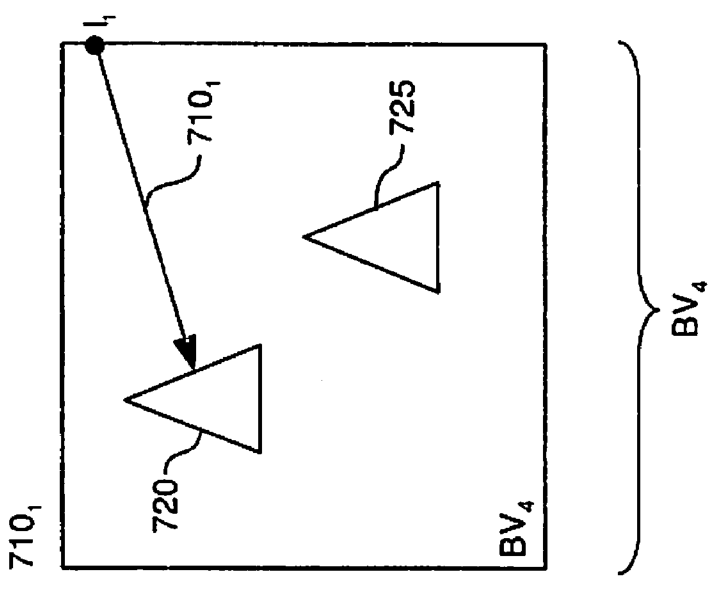

WORKLOAD MANAGER (WM)
205
CACHE
230
VTE
210
VECTOR TROUGHPUT ENGINE (VTE)
210
INBOXES
115
VTE
210

FIG. 8B


710_1
I_1
710_1
725
720
BV_4
BV_4

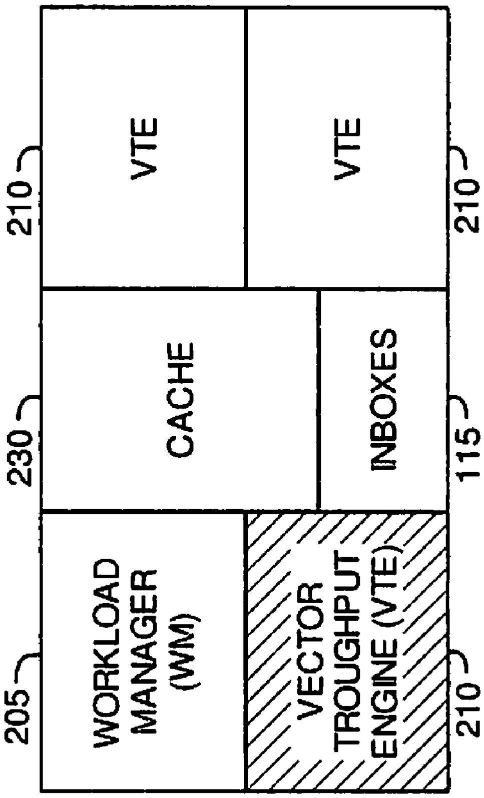
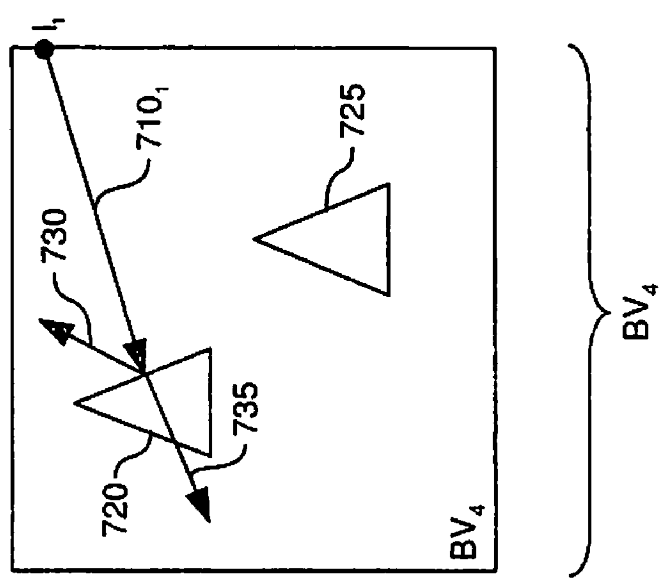

210
VTE
VTE
210
230
CACHE
INBOXES
115
205
WORKLOAD MANAGER (WM)
VECTOR TROUGHPUT ENGINE (VTE)
210

FIG. 8C


I_1
710_1
725
730
735
720
BV_4
BV_4

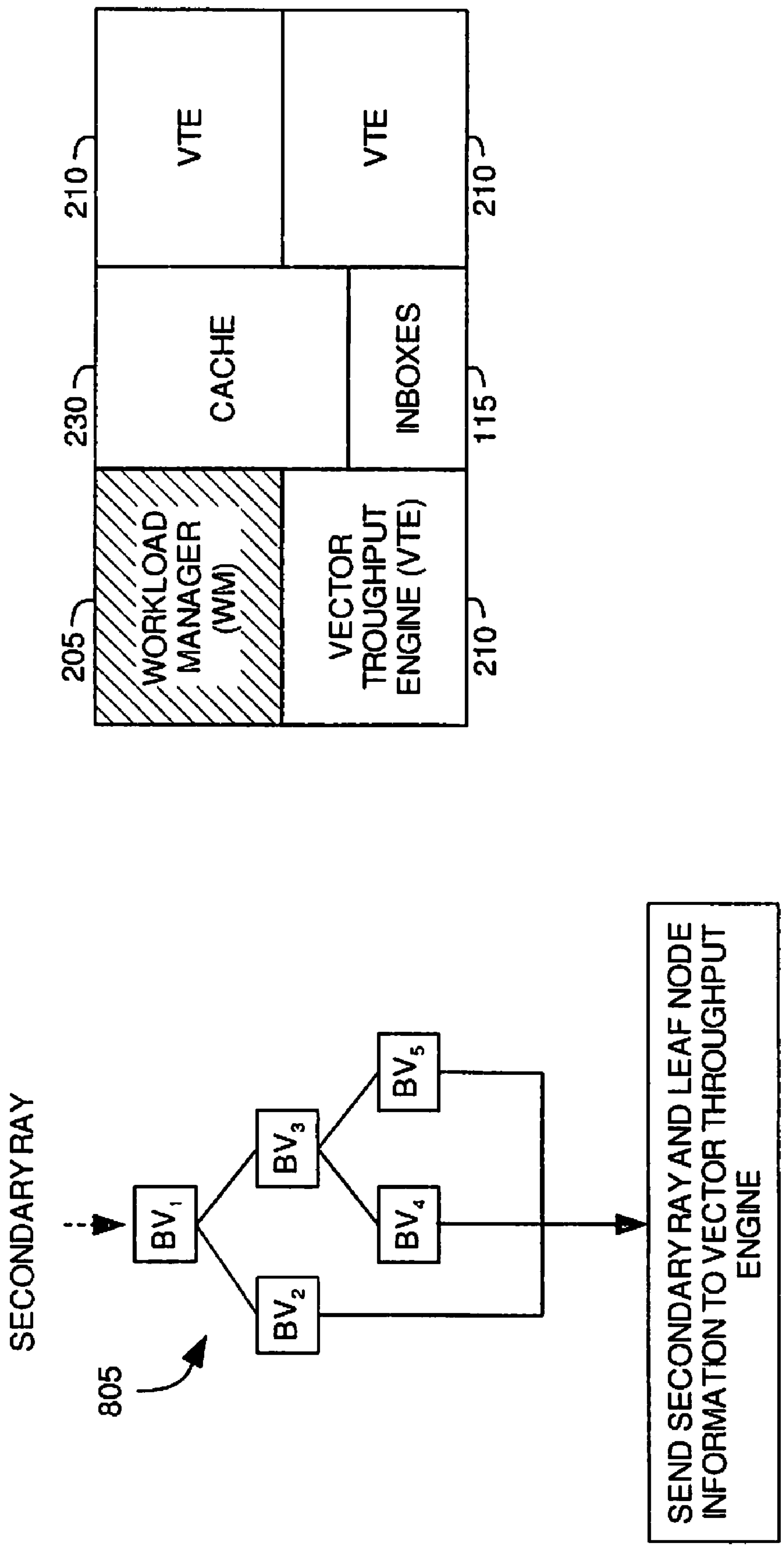

205
WORKLOAD MANAGER (WM)
230
CACHE
210
VTE
210
VECTOR TROUGHPUT ENGINE (VTE)
115
INBOXES
210
VTE
SECONDARY RAY
805
BV_1
BV_2
BV_3
BV_4
BV_5
SEND SECONDARY RAY AND LEAF NODE INFORMATION TO VECTOR THROUGHPUT ENGINE

FIG. 8D

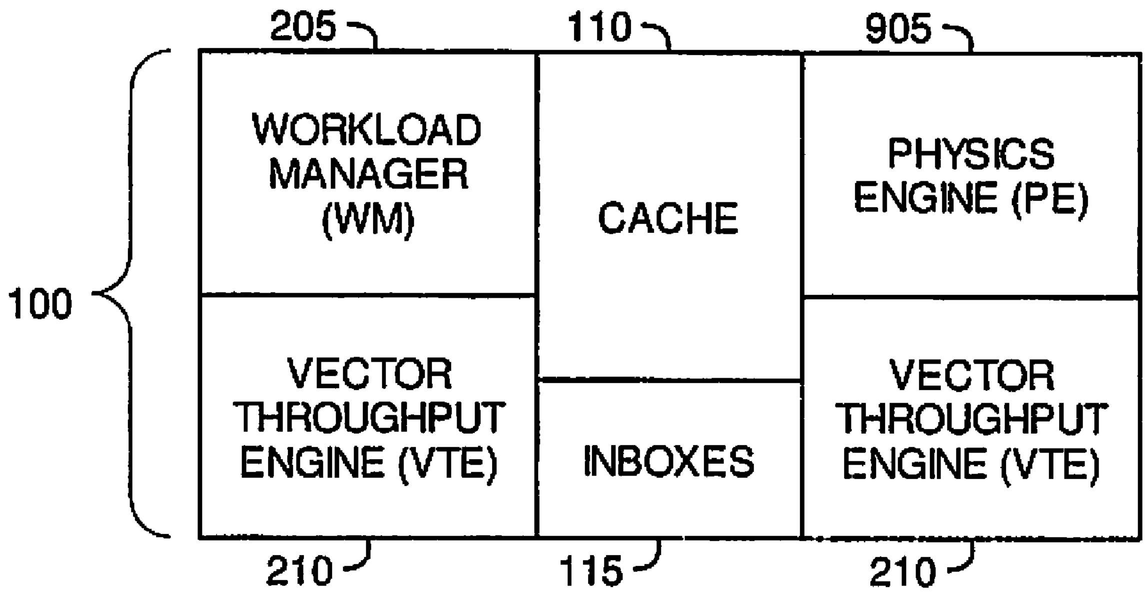

205
110
905
WORKLOAD MANAGER (WM)
CACHE
PHYSICS ENGINE (PE)
100
VECTOR THROUGHPUT ENGINE (VTE)
INBOXES
VECTOR THROUGHPUT ENGINE (VTE)
210
115
210

FIG. 9

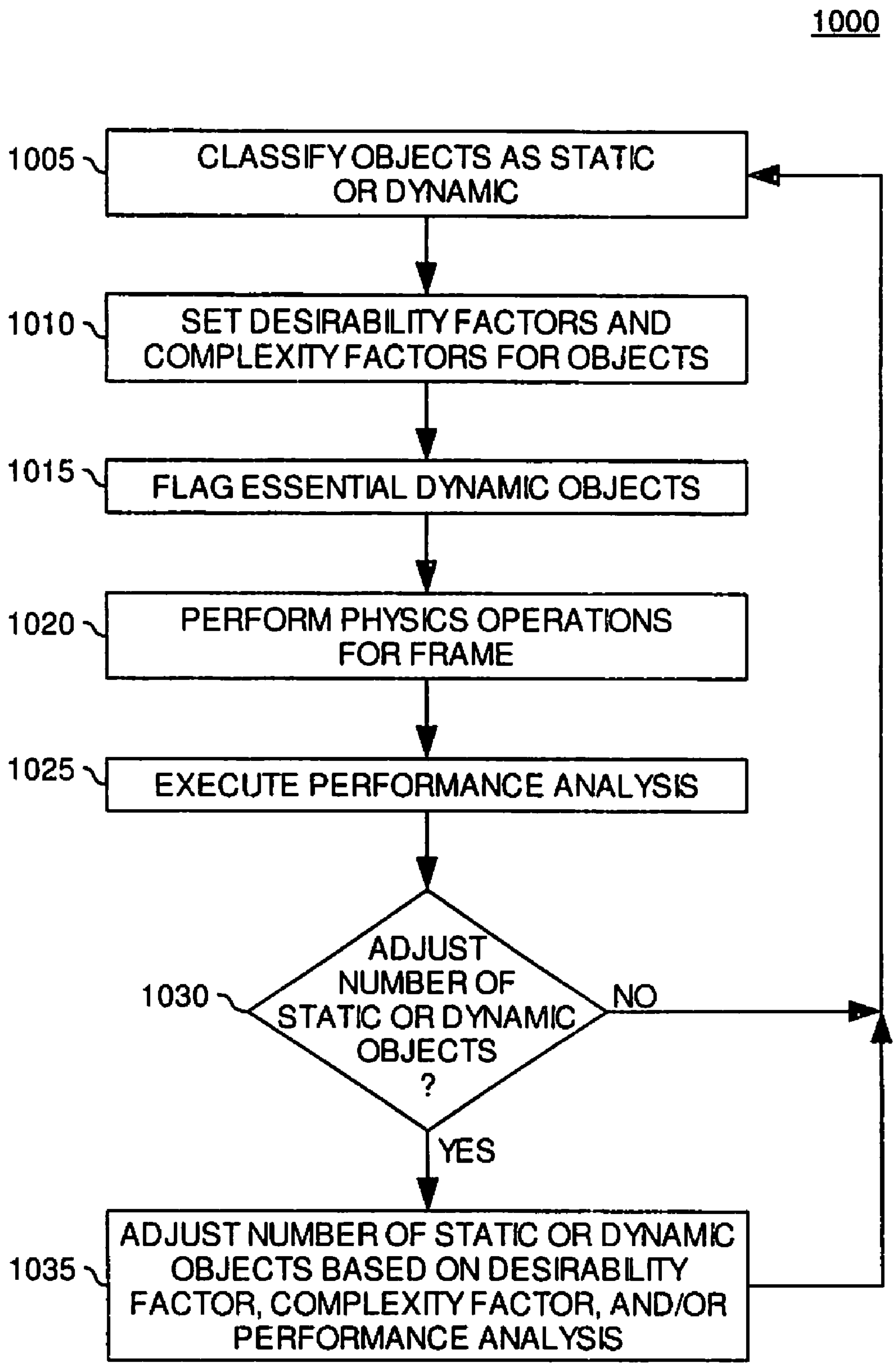

1000
1005
CLASSIFY OBJECTS AS STATIC OR DYNAMIC
1010
SET DESIRABILITY FACTORS AND COMPLEXITY FACTORS FOR OBJECTS
1015
FLAG ESSENTIAL DYNAMIC OBJECTS
1020
PERFORM PHYSICS OPERATIONS FOR FRAME
1025
EXECUTE PERFORMANCE ANALYSIS
1030
ADJUST NUMBER OF STATIC OR DYNAMIC OBJECTS ?
NO
YES
1035
ADJUST NUMBER OF STATIC OR DYNAMIC OBJECTS BASED ON DESIRABILITY FACTOR, COMPLEXITY FACTOR, AND/OR PERFORMANCE ANALYSIS

FIG. 10

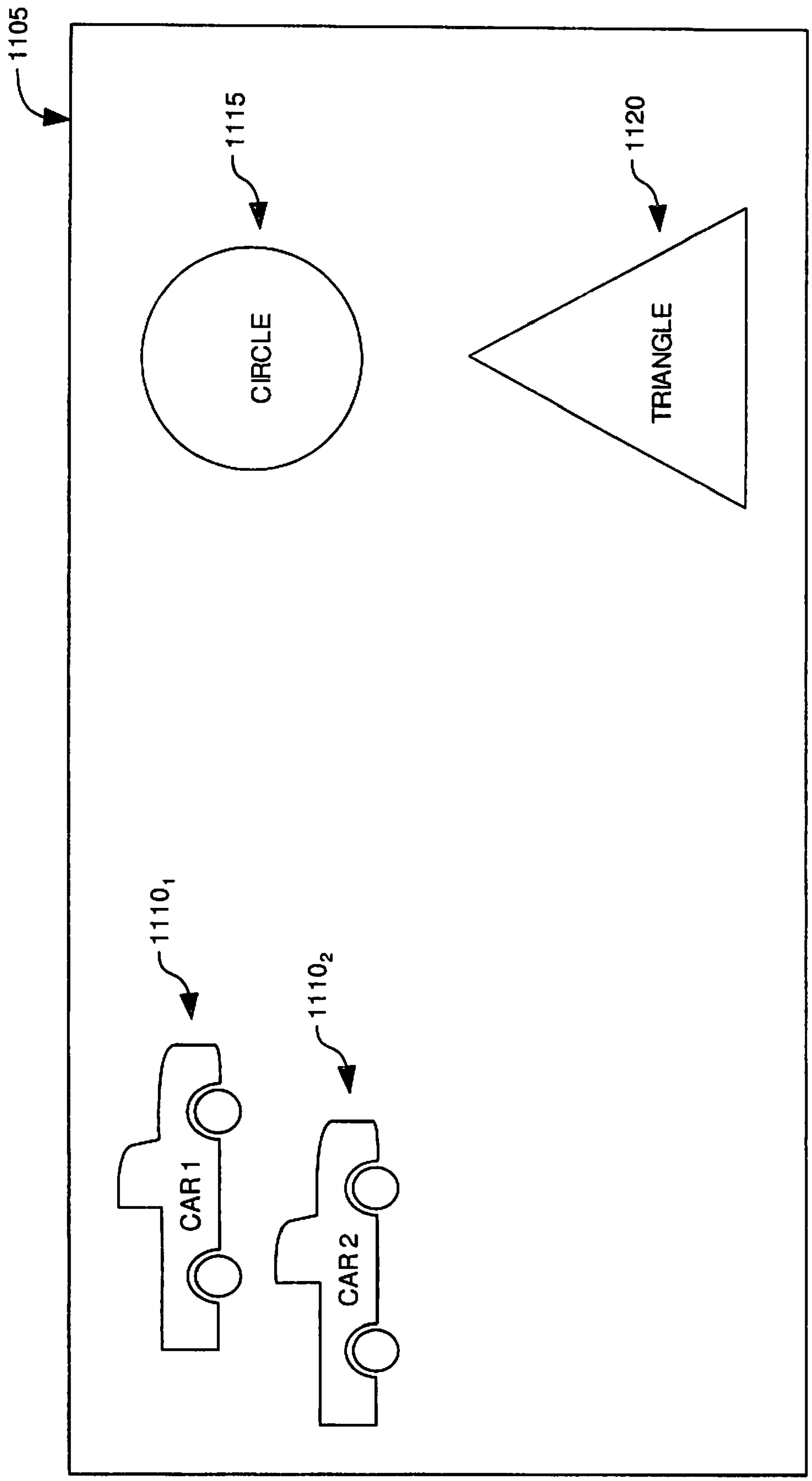


FIG. 11

1205

	DYNAMIC OBJECTS	DESIRABILITY FACTOR	COMPLEXITY FACTOR	ESSENTIAL OBJECT FLAG
1215	CAR1 OBJECT	8	7	1
1220	CAR2 OBJECT	8	7	0

1210

	STATIC OBJECTS	DESIRABILITY FACTOR	COMPLEXITY FACTOR
1225	CIRCLE OBJECT	4	3
1230	TRIANGLE OBJECT	3	3

FIG. 12

1205

DYNAMIC OBJECTS	DESIRABILITY FACTOR	COMPLEXITY FACTOR	ESSENTIAL OBJECT FLAG
1215 CAR1 OBJECT	8	7	1
1220 CAR2 OBJECT	8	7	0
1305 CIRCLE OBJECT	4	3	0

1210

STATIC OBJECTS	DESIRABILITY FACTOR	COMPLEXITY FACTOR
1230 TRIANGLE OBJECT	3	3

FIG. 13

1205

	DYNAMIC OBJECTS	DESIRABILITY FACTOR	COMPLEXITY FACTOR	ESSENTIAL OBJECT FLAG
1215	CAR1 OBJECT	8	7	1

1210

	STATIC OBJECTS	DESIRABILITY FACTOR	COMPLEXITY FACTOR
1225	CIRCLE OBJECT	4	3
1230	TRIANGLE OBJECT	8	3
1405	CAR2 OBJECT	8	7

FIG. 14

DYNAMICALLY LOAD BALANCING GAME PHYSICS USING REAL-TIME OBJECT SCALING

BACKGROUND OF THE INVENTION

1. Field of the Invention

Embodiments of the invention generally relate to the field of image processing.

2. Description of the Related Art

The process of rendering two-dimensional images from three-dimensional scenes is commonly referred to as image processing. As the modern computer industry evolves image processing evolves as well. One particular goal in the evolution of image processing is to make two-dimensional simulations or renditions of three-dimensional scenes as realistic as possible. One limitation of rendering realistic images is that modern monitors display images through the use of pixels.

A pixel is the smallest area of space which can be illuminated on a monitor. Most modern computer monitors will use a combination of hundreds of thousands or millions of pixels to compose the entire display or rendered scene. The individual pixels are arranged in a grid pattern and collectively cover the entire viewing area of the monitor. Each individual pixel may be illuminated to render a final picture for viewing.

One technique for rendering a real world three-dimensional scene onto a two-dimensional monitor using pixels is called rasterization. Rasterization is the process of taking a two-dimensional image represented in vector format (mathematical representations of geometric objects within a scene) and converting the image into individual pixels for display on the monitor. Rasterization is effective at rendering graphics quickly and using relatively low amounts of computational power; however, rasterization suffers from some drawbacks. For example, rasterization often suffers from a lack of realism because it is not based on the physical properties of light, rather rasterization is based on the shape of three-dimensional geometric objects in a scene projected onto a two-dimensional plane. Furthermore, the computational power required to render a scene with rasterization scales directly with an increase in the complexity of the scene to be rendered. As image processing becomes more realistic, rendered scenes also become more complex. Therefore, rasterization suffers as image processing evolves, because rasterization scales directly with complexity.

Another technique for rendering a real world three-dimensional scene onto a two-dimensional monitor using pixels is called ray tracing. The ray tracing technique traces the propagation of imaginary rays, rays which behave similar to rays of light, into a three-dimensional scene which is to be rendered onto a computer screen. The rays originate from the eye(s) of a viewer sitting behind the computer screen and traverse through pixels, which make up the computer screen, towards the three-dimensional scene. Each traced ray proceeds into the scene and may intersect with objects within the scene. If a ray intersects an object within the scene, properties of the object and several other contributing factors are used to calculate the amount of color and light, or lack thereof, the ray is exposed to. These calculations are then used to determine the final color of the pixel through which the traced ray passed.

The process of tracing rays is carried out many times for a single scene. For example, a single ray may be traced for each pixel in the display. Once a sufficient number of rays have been traced to determine the color of all of the pixels which make up the two-dimensional display of the computer screen, the two-dimensional synthesis of the three-dimensional scene can be displayed on the computer screen to the viewer.

Ray tracing typically renders real world three-dimensional scenes with more realism than rasterization. This is partially due to the fact that ray tracing simulates how light travels and behaves in a real world environment, rather than simply projecting a three-dimensional shape onto a two-dimensional plane as is done with rasterization. Therefore, graphics rendered using ray tracing more accurately depict on a monitor what our eyes are accustomed to seeing in the real world.

Furthermore, ray tracing also handles increases in scene complexity better than rasterization as scenes become more complex. Ray tracing scales logarithmically with scene complexity. This is due to the fact that the same number of rays may be cast into a scene, even if the scene becomes more complex. Therefore, ray tracing does not suffer in terms of computational power requirements as scenes become more complex as rasterization does.

Image processing systems (such as ray-tracing image processing systems) may be used in combination with a physics engine to provide animation in a three-dimensional scene. The physics engine may simulate real world physical phenomena as applied to objects within the three-dimensional scene. For example, the physics engine may perform position updates for a moving object, and may perform collision detection tests to determine if the object collides with any other objects within the three-dimensional scene.

One major drawback of game system using ray tracing image processing is the large number of calculations, and thus processing power, required to simulate the physics involved with a three-dimensional scene and to perform ray tracing to render the scene. This leads to problems when fast rendering is needed. For example, fast rendering may be necessary when a physics engine and an image processing system are to render graphics for animation in a game console. Due to the increased computational requirements for performing the physics calculations and to perform ray tracing it is difficult to render animation quickly enough to seem realistic (realistic animation is approximately twenty to twenty-four frames per second).

Therefore, there exists a need for more efficient techniques and devices to perform ray tracing and to perform physics simulation.

SUMMARY OF THE INVENTION

Embodiments of the present invention generally provide methods and apparatus for performing ray tracing and physics simulation.

According to one embodiment of the invention a method of varying physics workload is provided. The method generally comprising: classifying a plurality of objects within a three-dimensional scene as at least one of a static object or a dynamic object; performing physics based operations on the objects for a timeframe; analyzing workload experienced by processing elements during the timeframe; and based on the workload experienced by processing elements during the timeframe, modifying the classification of at least one of the objects within the three-dimensional scene.

According to another embodiment of the invention a computer readable medium is provided. The computer readable medium containing a program which, when executed, performs operations generally comprising: classifying a plurality of objects within a three-dimensional scene as at least one of a static object or a dynamic object; performing physics based operations on the objects for a timeframe; analyzing workload experienced by processing elements during the timeframe; and based on the workload experienced by processing elements during the timeframe, modifying the classification of at least one of the objects within the three-dimensional scene.

According to another embodiment of the invention a system is provided. The system generally comprising: a plurality of objects within a three-dimensional scene; a first processing element; and a second processing element configured to classify the plurality of objects as at least one of a static object or a dynamic object, perform physics based operations on the objects for a timeframe, analyze workload experienced by the first processing element during the timeframe, and, based on the workload experienced by the first processing element during the timeframe, modify the classification of at least one of the objects within the three-dimensional scene.

BRIEF DESCRIPTION OF THE DRAWINGS

FIGS. **1** and **9** illustrate block diagrams depicting exemplary computer processors, according to embodiments of the invention.

FIG. **2** illustrates a multiple-core processing element network, according to one embodiment of the invention.

FIGS. **3**A-**3**C are block diagrams illustrating aspects of memory inboxes according to one embodiments of the invention.

FIG. **4** is an exemplary three-dimensional scene to be rendered by an image processing system, according to one embodiment of the invention.

FIGS. **5**A-**5**C illustrate a two-dimensional space to be rendered by an image processing system and a corresponding spatial index created by an image processing system, according to one embodiment of the invention.

FIG. **6** is a flowchart illustrating a method of performing ray tracing, according to one embodiment of the invention.

FIGS. **7** and **11** are exemplary three-dimensional scenes to be rendered by an image processing system, according to embodiments of the invention.

FIGS. **8**A-**8**D illustrate a method of performing ray tracing, according to one embodiment of the invention.

FIG. **10** is a flowchart illustrating an exemplary method of varying physics workload, according to embodiments of the invention.

FIGS. **12**-**14** illustrate exemplary data queues, according to embodiments of the invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention provides methods and apparatus for dynamically varying a physics workload by scaling the classification of objects within a three-dimensional scene. According to embodiments of the invention, a physics engine may classify a plurality of objects within a three-dimensional scene as either static objects or as dynamic objects. The physics engine may then perform physics operations with the objects, and may monitor the workload experienced by processing elements within a system which includes the physics engine. Based on the workload experienced by the processing elements within the system, the physics engine may vary the number of objects which are classified as dynamic and vary the number of objects which are classified as static. By varying the classification of the objects, the physics engine may vary the workload experienced by the physics engine.

In the following, reference is made to embodiments of the invention. However, it should be understood that the invention is not limited to specific described embodiments. Instead, any combination of the following features and elements, whether related to different embodiments or not, is contemplated to implement and practice the invention. Furthermore, in various embodiments the invention provides numerous advantages over the prior art. However, although embodiments of the invention may achieve advantages over other possible solutions and/or over the prior art, whether or not a particular advantage is achieved by a given embodiment is not limiting of the invention. Thus, the following aspects, features, embodiments and advantages are merely illustrative and are not considered elements or limitations of the appended claims except where explicitly recited in a claim(s). Likewise, reference to "the invention" shall not be construed as a generalization of any inventive subject matter disclosed herein and shall not be considered to be an element or limitation of the appended claims except where explicitly recited in a claim(s).

One embodiment of the invention is implemented as a program product for use with a computer system such as, for example, the image processing system described below. The program(s) of the program product defines functions of the embodiments (including the methods described herein) and can be contained on a variety of computer-readable media. Illustrative computer-readable media include, but are not limited to: (i) information permanently stored on non-writable storage media (e.g., read-only memory devices within a computer such as CD-ROM disks readable by a CD-ROM drive); (ii) alterable information stored on writable storage media (e.g., floppy disks within a diskette drive or hard-disk drive); and (iii) information conveyed to a computer by a communications medium, such as through a computer or telephone network, including wireless communications. The latter embodiment specifically includes information downloaded from the Internet and other networks. Such computer-readable media, when carrying computer-readable instructions that direct the functions of the present invention, represent embodiments of the present invention.

In general, the routines executed to implement the embodiments of the invention, may be part of an operating system or a specific application, component, program, module, object, or sequence of instructions. The computer program of the present invention typically is comprised of a multitude of instructions that will be translated by the native computer into a machine-readable format and hence executable instructions. Also, programs are comprised of variables and data structures that either reside locally to the program or are found in memory or on storage devices. In addition, various programs described hereinafter may be identified based upon the application for which they are implemented in a specific embodiment of the invention. However, it should be appreciated that any particular program nomenclature that follows is used merely for convenience, and thus the invention should not be limited to use solely in any specific application identified and/or implied by such nomenclature.

An Exemplary Multiple Core Processing Element

FIG. **1** illustrates a multiple core processing element **100**, according to one embodiment of the invention. The multiple core processing element **100** includes a plurality of basic throughput engines **105** (BTEs). A BTE **105** may contain a plurality of processing threads and a core cache (e.g., an L1 cache). The processing threads located within each BTE may have access to a shared multiple core processing element memory cache **110** (e.g., a shared L2 cache).

The BTEs **105** may also have access to a plurality of inboxes **115**. The inboxes **115**, described further below with regards to FIG. **3**, may be memory mapped address space. The inboxes **115** may be mapped to the processing threads located within each of the BTEs **105**. Each thread located within the BTEs may have a memory mapped inbox and access to all of the other memory mapped inboxes **115**. The inboxes **115** make up a low-latency and high-bandwidth communications network used by the BTEs **105**.

The BTEs may use the inboxes **115** as a network to communicate with each other and redistribute data processing work amongst the BTEs. For some embodiments, separate outboxes may be used in the communications network, for example, to receive the results of processing by BTEs **105**. For other embodiments, inboxes **115** may also serve as outboxes, for example, with one BTE **105** writing the results of a processing function directly to the inbox of another BTE **105** that will use the results.

The aggregate performance of an image processing system may be tied to how well the BTEs can partition and redistribute work. The network of inboxes **115** may be used to collect and distribute work to other BTEs without corrupting the shared multiple core processing element cache **110** with BTE communication data packets that have no frame to frame coherency. An image processing system which can render many millions of triangles per frame may include many BTEs **105** connected in this manner.

In one embodiment of the invention, the threads of one BTE **105** may be assigned to a workload manager. An image processing system may use various software and hardware components to render a two-dimensional image from a three-dimensional scene. As described further below with regards to FIG. **6**, according to one embodiment of the invention, an image processing system may use a workload manager to traverse a spatial index with a ray issued by the image processing system. A spatial index, as described further below with regards to FIG. **5**, may be implemented as a tree type data structure used to partition a relatively large three-dimensional scene into smaller bounding volumes. An image processing system using a ray tracing methodology for image processing may use a spatial index to quickly determine ray-bounding volume intersections. In one embodiment of the invention, the workload manager may perform ray-bounding volume intersection tests by using the spatial index.

In one embodiment of the invention, other threads of the multiple core processing element BTEs **105** on the multiple core processing element **100** may be vector throughput engines. After a workload manager determines a ray-bounding volume intersection, the workload manager may issue (send), via the inboxes **115**, the ray to one of a plurality of vector throughput engines. According to one embodiment of the invention, and described further below with regards to FIG. **6**, the vector throughput engines may then determine if the ray intersects a primitive contained within the bounding volume. The vector throughput engines may also perform operations relating to determining the color of the pixel through which the ray passed.

FIG. **2** illustrates a network of multiple core processing elements **200**, according to one embodiment of the invention. FIG. **2** also illustrates one embodiment of the invention where the threads of one of the BTEs of the multiple core processing element **100** is a workload manager **205**. Each multiple core processing element $\mathbf{220}_{1\text{-}N}$ in the network of multiple core processing elements **200** may contain one workload manager $\mathbf{205}_{1\text{-}N}$, according to one embodiment of the invention. Each processor **220** in the network of multiple core processing elements **200** may also contain a plurality of vector throughput engines **210**, according to one embodiment of the invention.

The workload managers $\mathbf{220}_{1\text{-}N}$ may use a high speed bus **225** to communicate with other workload managers $\mathbf{220}_{1\text{-}N}$ and/or vector throughput engines **210** of other multiple core processing elements **220**, according to one embodiment of the invention. Each of the vector throughput engines **210** may use the high speed bus **225** to communicate with other vector throughput engines **210** or the workload managers **205**. The workload manager processors **205** may use the high speed bus **225** to collect and distribute image processing related tasks to other workload manager processors **205**, and/or distribute tasks to other vector throughput engines **210**. The use of a high speed bus **225** may allow the workload managers $\mathbf{205}_{1\text{-}N}$ to communicate without affecting the caches **230** with data packets related to workload manager **205** communications.

Low-Latency High-Bandwidth Communications Network

As described above, the aggregate performance of an image processing system may be tied to how well the BTEs can partition and redistribute work. According to one embodiment of the invention, memory space within a cache, referred to as a memory inbox, may be used to distribute work to a single processor thread. In an image processing system using a plurality of processors each having a plurality of threads, the collection of inboxes together may be referred to as a low-latency high-bandwidth communications network.

In multithreading processor such as a BTE **105**, a memory inbox may be assigned to a given thread (referred to herein as the owner thread). In one embodiment of the invention, the memory space for the inbox may be allocated from the shared memory cache **110** exclusively to the owner thread. By exclusively assigning the memory space in a cache to the owner thread, the owner thread may maintain enough memory space to cache its own instructions and data without other having other competing threads displace the owner thread's instructions and data. Thus, the memory inbox may improve execution of the owner thread by maintaining the owner thread's data and instructions in the assigned inbox portion of the cache and reducing the possibility of stalling the owner thread while data and instructions for the owner thread are retrieved from higher levels of memory. Furthermore, by assigning the memory space in a cache to the owner thread, data or instructions intended for the targeted thread may be stored only in an inbox allocated to the thread. Thus, data or instructions intended for the targeted thread are not stored throughout the shared memory cache **110**, rather only in the inbox allocated to the targeted thread.

Furthermore, the inbox memory may be used by other threads to efficiently communicate with the owner thread. For example, where another thread has data and/or instructions which are to be provided to the owner thread for an inbox, the other thread may send the data and/or instructions to the inbox where the data and/or instructions may be retrieved by the owner thread. Similarly, in some cases, the owner thread may use the inbox as an outbox to communicate information with other threads. For example, to communicate the information with another thread, the owner thread may place the information in the inbox and send a notification to the other thread indicating the location of the data and/or instructions, thereby allowing the other thread to retrieve the information. Optionally, the owner thread may provide the information directly to the inbox of the other thread. Thus, the inbox memory may be used to simplify communication between a sending and a receiving thread while preventing displacement of data and/or instructions being used by other threads.

FIG. **3**A is a block diagram of memory inboxes **302** . . . **318** in a multi-core processor element **100** according to one embodiment of the invention. The depiction of the memory inboxes **302** . . . **318** is intended to be a conceptual view and therefore is not limited to any particular physical configuration. As depicted, threads (e.g., threads T**0**-T**7**) executing in each core (e.g., the BTEs **105**) may have access to the shared L2 cache **110** via a shared L2 cache interface **322**. Furthermore, the L2 cache interface **322** may also be used by the threads T**0** . . . T**7** to access the corresponding memory inboxes **302** . . . **318**. As described above, in some cases, each inbox **302** . . . **318** may be assigned to a corresponding thread T**0**-T**7**. Thus, Inbox **0 302** may be assigned to thread T**0** and so on. As described below, by assigning a given inbox to a given thread, access to the assigned inbox may be unrestricted with respect to the owner thread while access by other threads may be restricted. Exemplary restrictions are described below in greater detail.

FIG. **3**B is a block diagram depicting the path of data from memory inboxes (e.g., inboxes **302** . . . **308**) and the shared L2 cache **110** transmitted to and from a processing core (e.g., BTE **105**). As described above, both the memory inboxes **302** . . . **308** and the shared L2 cache **110** may be accessed via the shared L2 cache interface **322**. Where a thread being executed in the BTE **105** retrieves data from an inbox **302** . . . **308** or from the shared L2 cache **110**, the retrieved data may be placed in the L1 cache **312** for the BTE **105**. Instructions for the thread may be issued from an issue unit **332**. In some cases, the BTE **105** may be configured to execute multiple threads concurrently. Thus, the issue unit **332** may be configured to issue instructions for multiple threads. In some cases, the BTE **105** may provide multiple execution units **334** . . . **338** which may be used to concurrently execute threads in the BTE **105**. The execution units **334** . . . **338** may include a fixed point execution unit **334**, a floating point execution unit **336**, and a branch execution unit **338**.

In some cases, a thread may update or produce data which is to be accessed later (e.g., by the same thread or by another thread). Where the updated data is to be accessed later, the thread may place the updated data in an L1 cache **312**. Furthermore, where desired, the updated data may also be placed in the L2 cache **110** or in an inbox **302** . . . **308** for the updating thread via the shared L2 cache interface **322**. In some cases, as described above, direct access to a given inbox (e.g., inbox **0 302**) via the shared L2 cache interface **322** may be limited to the thread (e.g., thread T**0**) which owns the given inbox.

In one embodiment of the invention, memory space within a memory inbox may be mapped to a global memory address (e.g., all levels of memory including the L1 cache **312**, L2 cache **110**, and main memory as well as all threads may use the same global memory address to access a given memory inbox). Thus, in one embodiment of the invention, to access the inbox memory space, the owner thread may merely read or write the desired information to a global memory address corresponding to the inbox memory space. A thread which does not own the memory inbox and which attempts to directly access the inbox via the global memory address, may have access to the inbox denied. Other forms of access may instead be provided to other non-owning threads, e.g., via packetized messages sent to the inbox.

Also, in one embodiment of the invention, information being stored in a memory inbox may not be cacheable. For example, while information in the L1 cache **312**, L2 cache **110**, and other memory level may be automatically cached by the multi core processing element **100** such that information requested from a given memory address may be automatically fetched from main memory and maintained in one of the cache levels **312**, **110** while being accessed. In contrast, the globally addressable memory in a given inbox may only be located in the inbox and may not be moved between different levels of the memory hierarchy (e.g., the main memory, the shared L2 cache memory **110** or the L1 cache memory) without being copied to a new address space outside of the inbox. Thus, accesses to an inbox by an owner thread may be performed quickly and directly to the inbox memory without waiting for information to be fetched from another level of the memory hierarchy and/or translated during fetching. The non-cacheability of inbox memory may also apply with respect to packetized access of the inbox described below. Furthermore, in an alternate embodiment of the invention, information stored in the inbox may be cached in other levels of the memory hierarchy.

Assignment of Memory Inboxes

In one embodiment of the invention, memory inboxes may be provided from the shared memory cache **110** (e.g., a portion of the L2 cache **110** may be reserved for the inbox memory **115**). FIG. **3**C is a block diagram depicting inbox memory **115** partitioned from the shared L2 cache **110** according to one embodiment of the invention.

As depicted, the size and location of each inbox **302**, **304**, etc. may be controlled by inbox control registers **340**. The status of each inbox **302**, **304**, etc. (e.g., enabled or disabled) may be indicated and/or modified via inbox status registers **362**. In one embodiment, access to the inbox control registers **340** may be unrestricted. Optionally, in some cases, access to the inbox control registers may be limited, for example, to a subset of approved threads (e.g., the owner thread, a parent of the owner thread, a specially designated control thread, and/or an operating system kernel thread). In one embodiment, the inbox control registers **340** may include a start address register **342**, **348** . . . **354**, a size register **344**, **350** . . . **356**, and an owner thread identification register **346**, **352** . . . **358**.

In one embodiment, the start address registers **342**, **348** . . . **354** may indicate a start address for each inbox **302**, **304**, etc. The size registers **344**, **350** . . . **358** may indicate the size of a corresponding inbox **302**, **304**, etc. The memory space for an inbox may thus occupy each address beginning from the corresponding start address and ranging through the indicated size of the inbox. The size may be indicated in any manner, for example, as an absolute size in bytes or as an integer multiple of a fixed size (e.g., the size in the size registers **344**, **350** . . . **358** may indicate the size in kilobytes).

In one embodiment, the owner thread identification register **346**, **352** . . . **358** may identify which thread (e.g., thread T**0**, T**1** . . . TN) owns a given inbox **302**, **304**, etc. While depicted with respect to threads and corresponding inboxes **1**, **2** . . . N, embodiment of the invention may be used with any type of thread and/or inbox identifier (e.g., a number, an address, etc.). In one embodiment of the invention, the inbox identifier register may be used to restrict direct access to memory addresses within the corresponding inbox to the owner thread. In some cases, direct access may also be allowed by a limited selection of other threads, such as, for example, a parent thread of the owner thread, a specified control thread, and/or an operating system kernel thread. In one embodiment, access control circuitry **360** may be used to provide the restricted access.

By assigning portions of the shared memory cache **110** to the inboxes a low-latency high-bandwidth communications network may be formed. The remaining portion of the shared memory cache **110** may remain unassigned and, thus, available to store information which does not relate to communications between processing threads. The remaining portion of the shared memory cache **110** may be used to store geometry and data structures which are used by the image processing system to perform ray tracing (described further below with respect to FIG. **5**).

A benefit of using only the inboxes for communications between processing threads and using the remaining portion of the shared memory cache **110** to store geometry and data structures is that no matter how much communications related information is passed through the inboxes, it will not consume the entire memory cache. Thus, as will be described further below, communications related information will not displace the geometry and data structures stored within the remaining portion of the shared memory cache **100**. Therefore, data which is likely to be reused when tracing subsequent rays or rendering subsequent frames (object geometry and data structures) may remain in the cache, while data which is unlikely to be reused when tracing subsequent rays or rendering subsequent frames (data processing work) will not remain in the cache.

An Exemplary Three-Dimensional Scene

FIG. **4** is an exemplary three-dimensional scene **405** to be rendered by an image processing system. Within the three-dimensional scene **405** may be objects **420**. The objects **420** in FIG. **4** are of different geometric shapes. Although only four objects **420** are illustrated in FIG. **4**, the number of objects in a typical three-dimensional scene may be more or less. Commonly, three-dimensional scenes will have many more objects than illustrated in FIG. **4**.

As can be seen in FIG. **4** the objects are of varying geometric shape and size. For example, one object in FIG. **4** is a pyramid $\mathbf{420}_A$. Other objects in FIG. **4** are boxes $\mathbf{420}_{B\text{-}D}$. In many modern image processing systems objects are often broken up into smaller geometric shapes (e.g., squares, circles, triangles, etc.). The larger objects are then represented by a number of the smaller simple geometric shapes. These smaller geometric shapes are often referred to as primitives.

Also illustrated in the scene **405** are light sources $\mathbf{425}_{A\text{-}B}$. The light sources may illuminate the objects **420** located within the scene **405**. Furthermore, depending on the location of the light sources **425** and the objects **420** within the scene **405**, the light sources may cause shadows to be cast onto objects within the scene **405**.

The three-dimensional scene **405** may be rendered into a two-dimensional picture by an image processing system. The image processing system may also cause the two-dimensional picture to be displayed on a monitor **410**. The monitor **410** may use many pixels **430** of different colors to render the final two-dimensional picture.

One method used by image processing systems to rendering a three-dimensional scene **420** into a two-dimensional picture is called ray tracing. Ray tracing is accomplished by the image processing system "issuing" or "shooting" rays from the perspective of a viewer **415** into the three-dimensional scene **420**. The rays have properties and behavior similar to light rays.

One ray **440**, that originates at the position of the viewer **415** and traverses through the three-dimensional scene **405**, can be seen in FIG. **4**. As the ray **440** traverses from the viewer **415** to the three-dimensional scene **405**, the ray **440** passes through a plane where the final two-dimensional picture will be rendered by the image processing system. In FIG. **4** this plane is represented by the monitor **410**. The point the ray **440** passes through the plane, or monitor **410**, is represented by a pixel **435**.

As briefly discussed earlier, most image processing systems use a grid **430** of thousands (if not millions) of pixels to render the final scene on the monitor **410**. Each individual pixel may display a different color to render the final composite two-dimensional picture on the monitor **410**. An image processing system using a ray tracing image processing methodology to render a two-dimensional picture from a three-dimensional scene will calculate the colors that the issued ray or rays encounters in the three-dimensional scene. The image processing scene will then assign the colors encountered by the ray to the pixel through which the ray passed on its way from the viewer to the three-dimensional scene.

The number of rays issued per pixel may vary. Some pixels may have many rays issued for a particular scene to be rendered. In which case the final color of the pixel is determined by the each color contribution from all of the rays that were issued for the pixel. Other pixels may only have a single ray issued to determine the resulting color of the pixel in the two-dimensional picture. Some pixels may not have any rays issued by the image processing system, in which case their color may be determined, approximated or assigned by algorithms within the image processing system.

To determine the final color of the pixel **435** in the two-dimensional picture, the image processing system must determine if the ray **440** intersects an object within the scene. If the ray does not intersect an object within the scene it may be assigned a default background color (e.g., blue or black, representing the day or night sky). Conversely, as the ray **440** traverses through the three-dimensional scene the ray **440** may strike objects. As the rays strike objects within the scene the color of the object may be assigned the pixel through which the ray passes. However, the color of the object must be determined before it is assigned to the pixel.

Many factors may contribute to the color of the object struck by the original ray **440**. For example, light sources within the three-dimensional scene may illuminate the object. Furthermore, physical properties of the object may contribute to the color of the object. For example, if the object is reflective or transparent, other non-light source objects may then contribute to the color of the object.

In order to determine the effects from other objects within the three-dimensional scene, secondary rays may be issued from the point where the original ray **440** intersected the object. For example, one type of secondary ray may be a shadow ray. A shadow ray may be used to determine the contribution of light to the point where the original ray **440** intersected the object. Another type of secondary ray may be a transmitted ray. A transmitted ray may be used to determine what color or light may be transmitted through the body of the object. Furthermore, a third type of secondary ray may be a reflected ray. A reflected ray may be used to determine what color or light is reflected onto the object.

As noted above, one type of secondary ray may be a shadow ray. Each shadow ray may be traced from the point of intersection of the original ray and the object, to a light source within the three-dimensional scene **405**. If the ray reaches the light source without encountering another object before the ray reaches the light source, then the light source will illuminate the object struck by the original ray at the point where the original ray struck the object.

For example, shadow ray $\mathbf{441}_A$ may be issued from the point where original ray **440** intersected the object $\mathbf{420}_A$, and may traverse in a direction towards the light source $\mathbf{425}_A$. The shadow ray $\mathbf{441}_A$ reaches the light source $\mathbf{425}_A$ without encountering any other objects **420** within the scene **405**.

Therefore, the light source 425_A will illuminate the object 420_A at the point where the original ray **440** intersected the object 420_A.

Other shadow rays may have their path between the point where the original ray struck the object and the light source blocked by another object within the three-dimensional scene. If the object obstructing the path between the point on the object the original ray struck and the light source is opaque, then the light source will not illuminate the object at the point where the original ray struck the object. Thus, the light source may not contribute to the color of the original ray and consequently neither to the color of the pixel to be rendered in the two-dimensional picture. However, if the object is translucent or transparent, then the light source may illuminate the object at the point where the original ray struck the object.

For example, shadow ray 441_B may be issued from the point where the original ray **440** intersected with the object 420_A, and may traverse in a direction towards the light source 425_B. In this example, the path of the shadow ray 441_B is blocked by an object 420_D. If the object 420_D is opaque, then the light source 425_B will not illuminate the object 420_A at the point where the original ray **440** intersected the object 420_A. However, if the object 420_D which the shadow ray is translucent or transparent the light source 425_B may illuminate the object 420_A at the point where the original ray **440** intersected the object 420_A.

Another type of secondary ray is a transmitted ray. A transmitted ray may be issued by the image processing system if the object with which the original ray intersected has transparent or translucent properties (e.g., glass). A transmitted ray traverses through the object at an angle relative to the angle at which the original ray struck the object. For example, transmitted ray **444** is seen traversing through the object 420_A which the original ray **440** intersected.

Another type of secondary ray is a reflected ray. If the object with which the original ray intersected has reflective properties (e.g. a metal finish), then a reflected ray will be issued by the image processing system to determine what color or light may be reflected by the object. Reflected rays traverse away from the object at an angle relative to the angle at which the original ray intersected the object. For example, reflected ray **443** may be issued by the image processing system to determine what color or light may be reflected by the object 420_A which the original ray **440** intersected.

The total contribution of color and light of all secondary rays (e.g., shadow rays, transmitted rays, reflected rays, etc.) will result in the final color of the pixel through which the original ray passed.

An Exemplary Kd-Tree

One problem encountered when performing ray tracing is determining quickly and efficiently if an issued ray intersects any objects within the scene to be rendered. One methodology known by those of ordinary skill in the art to make the ray intersection determination more efficient is to use a spatial index. A spatial index divides a three-dimensional scene or world into smaller volumes (smaller relative to the entire three-dimensional scene) which may or may not contain primitives. An image processing system can then use the known boundaries of these smaller volumes to determine if a ray may intersect primitives contained within the smaller volumes. If a ray does intersect a volume containing primitives, then a ray intersection test can be run using the trajectory of the ray against the known location and dimensions of the primitives contained within that volume. If a ray does not intersect a particular volume then there is no need to run ray-primitive intersection tests against the primitives contained within that volume. Furthermore, if a ray intersects a bounding volume which does not contain primitives then there is no need to run ray-primitive intersections tests against that bounding volume. Thus, by reducing the number of ray-primitive intersection tests which may be necessary, the use of a spatial index greatly increases the performance of a ray tracing image processing system. Some examples of different spatial index acceleration data structures are octrees, k dimensional Trees (kd-Trees), and binary space partitioning trees (BSP trees). While several different spatial index structures exist, for ease of describing embodiments of the present invention, a kd-Tree will be used in the examples to follow. However, those skilled in the art will readily recognize that embodiments of the invention may be applied to any of the different types of spatial indexes.

A kd-Tree uses axis aligned bounding volumes to partition the entire scene or space into smaller volumes. That is, the kd-Tree may divide a three-dimensional space encompassed by a scene through the use of splitting planes which are parallel to known axes. The splitting planes partition a larger space into smaller bounding volumes. Together the smaller bounding volumes make up the entire space in the scene. The determination to partition (divide) a larger bounding volume into two smaller bounding volumes may be made by the image processing system through the use of a kd-tree construction algorithm.

One criterion for determining when to partition a bounding volume into smaller volumes may be the number of primitives contained within the bounding volume. That is, as long as a bounding volume contains more primitives than a predetermined threshold, the tree construction algorithm may continue to divide volumes by drawing more splitting planes. Another criterion for determining when to partition a bounding volume into smaller volumes may be the amount of space contained within the bounding volume. Furthermore, a decision to continue partitioning the bounding volume may also be based on how many primitives may be intersected by the plane which creates the bounding volume.

The partitioning of the scene may be represented by a binary tree structure made up of nodes, branches and leaves. Each internal node within the tree may represent a relatively large bounding volume, while the node may contain branches to sub-nodes which may represent two relatively smaller partitioned volumes resulting after a partitioning of the relatively large bounding volume by a splitting plane. In an axis-aligned kd-Tree, each internal node may contain only two branches to other nodes. The internal node may contain branches (i.e., pointers) to one or two leaf nodes. A leaf node is a node which is not further sub-divided into smaller volumes and contains pointers to primitives. An internal node may also contain branches to other internal nodes which are further sub-divided. An internal node may also contain the information needed to determine along what axis the splitting plane was drawn and where along the axis the splitting plane was drawn.

Exemplary Bounding Volumes

FIGS. **5**A-**5**C illustrate a two-dimensional space to be rendered by an image processing system and a corresponding kd-tree. For simplicity, a two-dimensional scene is used to illustrate the building of a kd-Tree, however kd-Trees may also be used to represent three-dimensional scenes. In the two-dimensional illustration of FIGS. **5**A-**5**C splitting lines are illustrated instead of splitting planes, and bounding areas are illustrated instead of bounding volumes as would be used in a three-dimensional structure. However, one skilled in the art will quickly recognize that the concepts may easily be applied to a three-dimensional scene containing objects.

FIG. **5**A illustrates a two-dimensional scene **505** containing primitives **510** to be rendered in the final picture to be displayed on a monitor **510**. The largest volume which represents the entire volume of the scene is encompassed by bounding volume **1** (BV_1). In the corresponding kd-Tree this may be represented by the top level node **550**, also known as the root or world node. In one embodiment of an image processing system, an image processing system may continue to partition bounding volumes into smaller bounding volumes when the bounding volume contains, for example, more than two primitives. As noted earlier the decision to continue partitioning a bounding volume into smaller bounding volumes may be based on many factors, however for ease of explanation in this example the decision to continue partitioning a bounding volume is based only on the number of primitives. As can be seen in FIG. **5**A, BV_1 contains six primitives, therefore kd-Tree construction algorithm may partition BV_1 into smaller bounding volumes.

FIG. **5**B illustrates the same two-dimensional scene **505** as illustrated in FIG. **5**A. However, in FIG. **5**B the tree construction algorithm has partitioned BV_1 into two smaller bounding volumes BV_2 and BV_3. The partitioning of BV_1, was accomplished, by drawing a splitting plane SP_1 **515** along the x-axis at point x_1. This partitioning of BV_1 is also reflected in the kd-Tree as the two nodes **555** and **560**, corresponding to BV_2 and BV_3 respectively, under the internal or parent node BV_1 **550**. The internal node representing BV_1 may now store information such as, but not limited to, pointers to the two nodes beneath BV_1 (e.g., BV_2 and BV_3), along which axis the splitting plane was drawn (e.g., x-axis), and where along the axis the splitting plane was drawn (e.g., at point x_1).

The kd-Tree construction algorithm may continue to partition bounding volume BV_3 because it contains more than the predetermined threshold of primitives (e.g., more than two primitives). However, the kd-Tree construction algorithm may not continue to partition bounding volume BV_2, because bounding volume BV_2 contains less than or equal to the number of primitives (e.g., only two primitives $\mathbf{510}_A$). Nodes which are not partitioned or sub-divided any further, such as BV_2, are referred to as leaf nodes.

FIG. **5**C illustrates the same two-dimensional scene **505** as illustrated in FIG. **5**B. However, in FIG. **5**C the kd-Tree construction algorithm has partitioned BV_3 into two smaller bounding volumes BV_4 and BV_5. The kd-construction algorithm has partitioned BV_3 using a partitioning plane along the y-axis at point y_1. Since BV_3 has been partitioned into two sub-nodes it may now be referred to as an internal node. The partitioning of BV_3 is also reflected in the kd-Tree as the two leaf nodes **565** and **570**, corresponding to BV_4 and BV_5 respectively. BV_4 and BV_5 are leaf nodes because the volumes they represent are not further divided into smaller bounding volumes. The two leaf nodes, BV_4 and BV_5, are located under the internal node BV_3 which represents the bounding volume which was partitioned in the kd-Tree.

The internal node representing BV_3 may store information such as, but not limited to, pointers to the two leaf nodes (i.e., BV_4 and BV_5), along which axis the splitting plane was drawn (i.e., y-axis), and where along the axis the splitting plane was drawn (i.e., at point y_1).

The kd-Tree construction algorithm may now stop partitioning the bounding volumes because all bounding volumes located within the scene contain less than or equal to the maximum predetermined number of primitives which may be enclosed within a bounding volume. The leaf nodes may contain pointers to the primitives which are enclosed within the bounding volumes each leaf represents. For example, leaf node BV_2 may contain pointers to primitives $\mathbf{510}_A$, leaf node BV_4 may contain pointers to primitives $\mathbf{510}_B$, and leaf node BV_5 may contain pointers to primitives $\mathbf{510}_C$.

The resulting kd-Tree structure, or other spatial index structure, may be stored in the shared memory cache **110**. The kd-Tree and the size of corresponding data which comprises the kd-Tree may be optimized for storage in the shared memory cache **110**.

Iterative Ray Tracing Algorithm

According to one embodiment of the invention, transforming the ray tracing algorithm from a recursive algorithm into an iterative algorithm may enable efficient distribution of workload related to ray tracing amongst a plurality of processing elements. An iterative ray tracing algorithm, in contrast to a recursive ray tracing algorithm, may allow separate processing elements to perform operations relating to determining the color of a single pixel and allow efficient use of processor resources (e.g., memory cache). Efficient distribution of workload amongst a plurality of processing elements may improve ray tracing image processing system performance.

An algorithm for performing ray tracing may be recursive in the sense that it issues an original ray into a three-dimensional scene and finishes all ray tracing operations relating to the issued original ray (e.g., traces all secondary rays and performs all ray-object intersection tests) before issuing a subsequent original ray into the three-dimensional scene.

For example, an image processing system may use a recursive ray tracing algorithm to render a two-dimensional image from a three-dimensional scene. The image processing system using a recursive ray tracing algorithm may use a processing element to perform ray tracing. The processor may be used to traverse a ray through a spatial index, and to determine if the ray intersects any objects within a bounding volume of the spatial index. If the ray intersects an object contained within a bounding volume, the image processing system, using the same processor, may issue secondary rays into the three-dimensional scene to determine if they intersect any objects and, consequently, contribute color to the object intersected by the original ray. While performing operations related to determining if the secondary rays intersect objects within the three-dimensional scene, the processor may store information defining the original ray in the processor's memory cache.

If the processing element determines that the secondary rays intersect objects within the three-dimensional scene the image processing element may issue more secondary rays into the scene to determine if those secondary rays intersect objects and contribute color to the object intersected by the original ray. When performing calculations to determine if the secondary rays intersect objects within the three-dimensional scene, the processor may store previous secondary ray information in the processor's memory cache. By issuing more and more secondary rays into the scene, the image processing system may finally determine the total contribution of color from secondary rays to the object intersected by the original ray. From the color of the object intersected by the original ray and the contribution of color due to secondary rays, the color of the pixel through which the original ray passed may be finally determined.

Although the recursive ray tracing algorithm determines the color of the pixel through which the original ray passed, each time the image processing system issues more secondary rays into the three-dimensional scene, the recursive ray tracing image processing system places information which defines the previous rays (e.g., the original ray or previous secondary rays) into the memory cache of the processing element. The image processing system may store ray information in the cache in order to free registers which may be necessary to perform the calculations related to determining if the subsequent secondary rays intersect objects within the three-dimensional scene. Consequently, the recursive ray tracing image processing system may place a large (relative to the size of the cache) amount of information into the processors memory cache for a single pixel.

By storing large amounts of ray information in the memory cache of the processor, there is little or no space in the processor's memory cache for information which defines the objects within the three-dimensional scene (i.e., object geometry data). This information may need to be frequently fetched from main memory into the memory cache in order to perform operations to determine if the original or secondary rays intersect objects within the three-dimensional scene (thereby "thrashing" the cache). Therefore, the limits of an image processing system which uses the recursive ray tracing technique may be limited by the access time to fetch information from main memory and place it in the processor's memory cache.

However, according to embodiments of the invention, the ray tracing algorithm may be partitioned into an iterative ray tracing algorithm. The iterative ray tracing algorithm may allow separate processing elements to perform portions of the ray tracing algorithm. By allowing separate processing elements to perform portions of the ray tracing algorithm, the amount of information which needs to be cached (e.g., original rays and secondary rays) may be reduced. Furthermore, according to embodiments of the invention, the iterative ray tracing algorithm may be used in conjunction with the low-latency high-bandwidth communications network and the shared memory cache **110** in order to improve the performance of a ray tracing image processing system.

The low-latency high-bandwidth communications network of inboxes, as described above with regards to FIGS. **3**A-**3**C, may be used to pass or send data processing information (e.g., information defining original rays and secondary rays) which has little use when tracing subsequent rays or rendering subsequent frames, according to embodiments of the invention. In addition, according to embodiments of the invention, the ray tracing image processing system may use a shared coherent memory cache to store information which may be used by the image processing system when tracing subsequent rays or performing ray tracing for a subsequent frame.

FIG. **6** is a flowchart which illustrates a partitioned and thus iterative ray tracing algorithm or method **600** which may be used in a multi processor image processing system, according to one embodiment of the invention. The method **600** begins at step **605** when the image processing system issues an original ray into the three-dimensional scene. The original ray may pass through a pixel as it traverses into the three-dimensional scene. The original ray may be used to determine the color of the pixel through which the original ray passed.

Next, at step **610** the image processing system may use a use a workload manager **205** processing element to traverse the spatial index (e.g., kd-Tree). The spatial index may be stored within the shared memory cache **110** of the image processing system. Traversing the kd-Tree may include performing calculations which determine if the original ray intersects bounding volumes which are defined by nodes within the spatial index. Furthermore, traversing the spatial index may include taking branches to nodes which defined bounding volumes intersected by the ray. A workload manager **205** may use the coordinates and trajectory of an issued ray (e.g., the original ray) to determine if the ray intersects bounding volumes defined by the nodes in the spatial index. The workload manager **205** may continue traversing the spatial index until the original ray intersects a bounding volume which contains only primitives (i.e., a leaf node).

At step **615**, after the workload manager **205** has traversed the original ray to a leaf node, the workload manager **205** may send the original ray and information which defines the leaf node to a vector throughput engine **210**. The workload manager **205** may send information which defines the original ray and the leaf node (e.g., trajectory of the ray, pixel through which the original ray passed, bounding volume defined by the leaf node, etc.) to the vector throughput engine **210**. The workload manager **205** may send the information to the vector throughput engine **210** by writing the information defining the ray and the intersected leaf node to the inbox of the vector throughput engine **210**.

By coupling the pixel information with the information which defines the original ray, there is no need to send the original ray back to the workload manager **205** if the vector throughput engine **210** determines that the ray intersected an object and, consequently, determines a color of the pixel. According to one embodiment of the invention, the vector throughput engine **210** may use the pixel information to update the color of the pixel by writing to memory location within a frame buffer (e.g., stored in the shared memory cache **110**) which corresponds to the pixel. By updating the pixel color as secondary rays intersect objects within the three-dimensional scene, the number of rays relating to the same pixel that need to be stored (e.g., in cache memory) may be reduced.

After the workload manager **205** sends the original ray information to the vector throughput engine **210**, the image processing system may issue a subsequent original ray into the three-dimensional scene. The workload manager **205** may immediately begin traversing this subsequently issued original ray through the spatial index after the workload manager **205** has sent the original ray to a vector throughput engine **210**. Thus, the workload manager **205** may be continuously traversing rays through the spatial index, rather than wait until the determination of whether the original ray intersected an object is complete, as in a recursive ray tracing algorithm. Furthermore, the workload manager **205** may be traversing rays through the spatial index as the vector throughput engine **210** is determining if previously issued rays intersect objects within the bounding volumes defined by leaf nodes. According to one embodiment of the invention, vector throughput engines **210** may be responsible for performing ray-primitive intersection tests. That is, the vector throughput engines **210** may determine if a ray intersects any primitives contained within the bounding volume defined by the leaf node.

Therefore, at step **620**, a vector throughput engine **210** that receives the ray and leaf node information in its inbox may perform ray-primitive intersection tests to determine if the ray intersects any primitives within the bounding volume defined by the leaf node. The geometry which defines the primitives may be stored within the shared memory cache **110**, and thus may not need to be fetched from main memory. By storing the geometry for primitives in the shared memory cache **110**, the iterative ray tracing algorithm may not need to fetch the geometry from main memory as is the case with the recursive ray tracing algorithm. If the vector throughput engine **210** determines that the original ray intersected a primitive contained within the bounding volume defined by the leaf node, the vector throughput engine **210** may proceed to step **630**.

At step **630**, the vector throughput engine **210** may determine the color of the intersected primitive at the point which the original ray intersected the primitive. For example, the color of the primitive may be stored in the shared memory cache **110** and the vector throughput engine **210** may read the color information from the shared memory cache **210**.

After determining the color of the primitive at the ray-primitive intersection point, the vector throughput engine **210** may update the color of pixel through which the ray passed. This may be accomplished, for example, by writing to a memory location within a frame buffer which corresponds to the pixel through which the original ray passed. By updating the pixel information as a ray-primitive intersection is determined and before determining the color contributions for all secondary rays relating to a original ray, the amount of information which may need to be stored in a memory cache may be reduced. In contrast, a recursive ray tracing algorithm may not store the color of the pixel in a frame buffer until all color contributions from secondary rays have been determined, which increases the amount of information which may need to be stored in a processor's memory cache.

After updating the pixel color, the vector throughput engine **210** may proceed to step **635**, where, the vector throughput engine **210** may generate secondary rays. As described previously with regards to FIG. **4**, a ray tracing image processing system may use secondary rays determine additional color contribution to the intersected object and thus to the pixel through which the original ray passed. Secondary rays may be, for example, reflected rays, transmitted (refracted) rays, or shadow rays. Generating secondary rays may include, for example, determining the trajectory of the secondary rays based on the trajectory of the original ray, surface properties of the intersected object, and an angle of intersection of the original ray with the intersected object.

After generating secondary rays, the vector throughput engine **210**, at step **640** may send the secondary rays to a workload manager **205**. The vector throughput engine **210** may send the secondary rays to a workload manager **205** by placing the information which defines the secondary rays (e.g., trajectory, information defining the pixel through which the original ray passed, etc.) in an inbox **115** of a workload manager **205**. According to one embodiment of the invention, the vector throughput engine **210** may send the secondary rays to the workload manager **205** which traversed the original ray through the spatial index. However, according to another embodiment of the invention, the image processing system may contain a plurality of workload managers and the vector throughput engine **210** may send the secondary rays to a workload manager which did not traverse the original ray through the spatial index.

After sending the secondary rays to a workload manager **205**, the vector throughput engine **210** may retrieve other information defining rays from an inbox which may be waiting to have ray-primitive intersection tests performed. The rays waiting in the vector throughput engine's **210** inbox may have been previously traversed through a spatial index by a workload manager **205**. Therefore, the vector throughput engine **210** may perform more ray-primitive intersection tests to determine if rays (i.e., original or secondary) intersect objects within bounding volumes defined by leaf nodes. Thus, the vector throughput engine **210** may continuously perform operations related to ray-primitive intersection tests, determining primitive colors, updating pixel colors, and generating secondary rays.

After receiving a secondary ray from a vector throughput engine **210**, a workload manager **205** may execute steps **610** and **615**, as described above, to determine if the secondary ray intersects a leaf node.

Returning to step **625**, if the vector throughput engine **210** determines that the ray did not intersect a primitive contained within bounding volume defined by the leaf node, the vector throughput engine **210** may assign the pixel through which the original ray passed a background color of the three-dimensional scene. The background color may be assigned to the pixel because the original ray did not intersect any primitives contained within the three-dimensional scene. However, according to other embodiments of the invention, if the ray did not intersect any primitives contained within the leaf-node bounding volume, the vector throughput engine **210** may send the ray back to a workload manager **205** such that the workload manager **205** may traverse the ray through the spatial index again to determine if the ray intersected any other leaf nodes containing primitives.

Exemplary Use of an Iterative Ray Tracing Algorithm

FIG. **7** illustrates exemplary rays issued from an image processing system into a three-dimensional scene **505**, according to one embodiment of the invention. For clarity, the three-dimensional scene **505** is the same as the three-dimensional scene used in FIGS. **5A-5C** to illustrate the construction of a kd-tree. Therefore, the kd-tree which corresponds to the three-dimensional scene **505** is the same as the kd-tree which was constructed with regards FIGS. **5A-5C**. As illustrated in FIG. **7**, a viewer **705** represents the origin of a plurality of original rays $\mathbf{710}_{1\text{-}4}$ which may be issued into the three-dimensional scene **505** by the image processing system. As each original ray $\mathbf{710}_{1\text{-}4}$ is issued into the three-dimensional scene, the original rays may first pass through a corresponding pixel in a grid (frame) of pixels **715**. Although only four pixels **715** and four original rays $\mathbf{710}_{1\text{-}4}$ are illustrated in FIG. **7**, to render a final two-dimensional image from a three-dimensional scene many more pixels may be necessary, and many more original rays may be issued.

A first original ray $\mathbf{710}_1$ may be issued by the image processing system and pass through a first pixel $\mathbf{715}_1$. The first original ray $\mathbf{710}_1$ may intersect bounding volume **4** (BV_4) at an intersection point I_1. To facilitate understanding, the image processing system in this example may follow a pattern of issuing rays starting from the top of the grid of pixels **715** and continue issuing rays, one ray per pixel, moving down the grid of pixels until a ray has been issued for each pixel in the grid of pixels.

A second original ray $\mathbf{710}_2$ and a third original ray $\mathbf{710}_3$ may also be issued by the image processing system which may pass through a second pixel $\mathbf{715}_2$ and a third pixel $\mathbf{715}_3$ respectively. The second original ray $\mathbf{710}_2$ and the third original ray $\mathbf{710}_3$ may also intersect BV_4 at a second intersection point I_2 and a third intersection point I_3, respectively. Thus the first original ray $\mathbf{710}_1$, the second original ray $\mathbf{710}_2$, and the third original ray $\mathbf{710}_3$ all intersect the same bounding volume. Furthermore, a fourth original ray $\mathbf{710}_4$ may be issued by the image processing system and may pass through a fourth pixel $\mathbf{815}_4$. The fourth original ray $\mathbf{710}_4$, in contrast to the first three original rays $\mathbf{710}_{1\text{-}3}$, may intersect bounding volume **5** (BV_5) at intersection point I_4.

FIG. **8A** illustrates the traversal of the first original ray $\mathbf{710}_1$ ray through a spatial index **805** (e.g., a kd-tree). Furthermore, as indicated by the shaded box **205**, FIG. **8A** illustrates a workload manager **205** performing operations related to the traversal of the first original ray 710_1 through the spatial index **805**. The workload manager **205** may traverse the ray through the spatial index **805** by taking branches to nodes defining bounding volumes intersected by the ray until a leaf node is reached (as illustrated in FIG. **8**A by the darkened branches and nodes). As illustrated in FIG. **7** the original ray 710_1 intersects BV_4, therefore, the workload manager **205** will traverse the first original ray 710_1 to the leaf node which defines BV_4. After traversing the ray to a leaf node, the workload manager **205** may send the first original ray 710_1 (e.g., send information which defines the first original ray 710_1 and information which defines the pixel 715_1 through which the first original ray passed) and information defining the intersected leaf node (i.e., BV_4) to a vector throughput engine **210**.

According to embodiments of the invention, after the workload manager **205** sends the first original ray 710_1 to a vector throughput engine **210**, the workload manager **205** may begin traversing the second original ray 710_2 through the spatial index. Thus, the workload manager **205** may be constantly traversing rays through the spatial index **805** while the vector throughput engines **210** are determining if rays intersect objects within the bounding volumes defined by traversed to leaf nodes.

FIG. **8**B illustrates the first original ray 710_1 traversing through the bounding volume **4** (BV**4**). Furthermore, as indicated by the shaded box, FIG. **8**B illustrates the vector throughput engine **210** performing ray-primitive intersection tests after the vector throughput engine has received the information defining the first original ray 710_1 and the information defining the bounding volume BV_4. As described with regards to FIG. **6**, the vector throughput engine **210** may execute ray-primitive intersection tests to determine if the original ray 710_1 intersects primitives contained within the bounding volume BV_4.

The vector throughput engine **210** may perform tests with the first original ray 710_1 against a first object **720** within the bounding volume BV_4, and against a second object **725** within the bounding volume BV_4. As illustrated in FIG. **8**B, the vector throughput engine **210** may determine that the first original ray 710_1 intersects the first object **720**.

As described previously with respect to method **600**, after determining that the first original ray 710_1 intersects an object, the vector throughput engine **210** may determine the color of the first object **720** at the point which the first original ray 710_1 intersected the first object **720**. After determining the color of the object **720** at the intersection point, the vector throughput engine **210** may update the color of the pixel 715_1 through which the first original ray 710_1 passed (e.g., by writing to a frame buffer memory location which corresponds to the pixel 715_1).

After determining the color of the object **720** at the intersection point, the vector throughput engine **210** may generate secondary rays. For example, as illustrated in FIG. **8**C the vector throughput engine **210** may generate a reflected ray **730** and a transmitted (refracted) ray **735**. Both secondary rays (**730** and **735**) originate from the point where the first original ray 710_1 intersected the object **720**. As described above, the secondary rays may be used to determine additional color contribution to the object at the point which the first original ray 710_1 intersected the object **720**. The generation of the secondary rays may include determining a trajectory for each secondary ray and tagging the secondary ray such that the additional color contribution from the secondary ray may be used to update the color of the pixel 715_1 through which the first original ray 710_1 passed.

After generating the secondary rays (**730** and **735**), the vector throughput engine **210** may send the secondary rays (**730** and **735**), via an inbox, to a workload manager **205**. A workload manager **205** which receives the secondary rays (**730** and **735**) may use the information which defines the secondary rays (i.e., trajectory of secondary rays) to traverse the spatial index **805**. For example, the shaded box in FIG. **8**D illustrates a workload manager **205** which may traverse the spatial index **805** with a secondary ray (e.g., **730**) which was generated by a vector throughput engine **210**. The workload manager **205** may traverse the secondary ray to a leaf node. After the secondary ray has been traversed to a leaf node, the workload manager **205** may send the secondary ray and information defining the bounding volume intersected by the secondary ray to a vector throughput engine **210** to determine if the secondary ray intersects any objects with the bounding volume intersected by the secondary ray.

As the vector throughput engines **210** determine that the original ray or secondary rays strike objects within the three-dimensional scene, the color of the pixel through which the original ray passed may be updated within the frame buffer. According to embodiments of the invention, all secondary rays relating to an original ray, and thus to the pixel through which the original ray passed, may be traced through the three-dimensional scene and their color contributions saved in the frame buffer to determine the final color of the pixel. However, according to other embodiments of the invention, a finite number of secondary rays relating to the original ray may be traced through the three-dimensional scene to determine the color of the pixel. By limiting the number of secondary rays which are traced through the three-dimensional scene and thus contribute to the color of the pixel, the amount of processing necessary to determine a final color of the pixel may be reduced.

Physics Engine

A physics engine is an application which may simulate real world physical phenomena as applied to objects within a three-dimensional scene. A physics engine may be used to simulate and predict the effects of physical phenomena on a frame to frame basis. For example, the physics engine may perform position updates for an object if the object is moving, and may perform collision detection tests to determine if an object collides with any other objects within the three-dimensional scene.

An image processing system may be used in conjunction with a physics engine to render the simulated physical interactions and objects within a three-dimensional scene to a two-dimensional screen. For example, a video game engine may use both a physics engine and an image processing system to simulate object movements or interactions within a three-dimensional scene and to display the objects and the environment on a monitor.

According to one embodiment of the invention, a physics engine may use multiple threads on a multiple core processing element to perform multiple physics related calculations in parallel. For example, FIG. **9** illustrates a multiple core processing element **100** wherein the threads of one of the cores are allocated to a physics engine **905**. Other cores within the multiple-core processing element may perform image processing related tasks, according to embodiments of the invention. For example, one core within the multiple-core processing element **100** may be allocated to a workload manager **205** and other cores within the multiple-core processing element **100** may be allocated to vector throughput engines **210**, according to one embodiment of the invention.

The multiple-core processing element **100** may have a memory cache **110** shared between all of the cores located on the multiple-core processing element **100**. Furthermore, each core may have its own cache (e.g., an L1 cache). The multiple-core processing element **100** may also contain inboxes **115**. The inboxes **115** may be memory mapped address space used by the cores as a communications network.

Dynamically Load Balancing Game Physics Using Real-Time Object Scaling

In some circumstances the image processing system (e.g., a ray-tracing image processing system) may be used in conjunction with a physics engine to provide animation. For example, a game system may use a physics engine and a ray-tracing image processing system to provide animation. As stated above, a physics engine may utilize multiple threads within a multiple core processing element. Furthermore, in some circumstances the amount of processing bandwidth required to perform physics operations may vary from frame to frame.

Some of the more computationally intensive operations a physics engine must perform are object collision operations. These operations may include detecting collisions between moving objects and other objects and performing operations to handle the aftermath of a collision. Objects which are moving within the scene may be classified as dynamic objects, while objects which are not moving may be classified as static. Dynamic objects require more operations when performing collision operations. Therefore, the more dynamic objects within a three-dimensional scene, the more physics operations which may be performed by the physics engine. In contrast, the more objects in a scene which are static, the fewer operations to be performed by the physics engine. Consequently, the number of dynamic objects within a three-dimensional scene is directly proportional to the physics operations workload to be performed by a physics engine.

Embodiments of the invention provide methods and apparatus to dynamically vary the amount of physics workload based on the amount of processing bandwidth available (e.g., the number of processing threads available) for physics based tasks. Embodiments of the invention may vary the amount of physics workload by dynamically scaling or adjusting the classification of objects within the scene as static objects or as dynamic objects.

FIG. **10** is a flowchart illustrating an exemplary method **1000** of varying the amount of physics workload, according to one embodiment of the invention. The method **1000** may begin at step **1005**, for example, when a game system begins to use a physics engine. Initially, at step **1005**, the physics engine may arbitrarily classify the objects within a three-dimensional scene as static or dynamic. The physics engine may classify objects by creating an entry for each object within the three-dimensional scene in either a static object queue or a dynamic object queue.

For example, a game system may simulate driving a car within a three-dimensional scene. FIG. **11** is an exemplary three-dimensional scene **1105** which may be used by a game system to simulate driving. The three-dimensional scene **1105** contains several objects including a first car object **1110**$_1$, a second car object **1110**$_2$, a circle object **1115**, and a triangle object **1120**. The physics engine may simulate real world physical phenomena as applied to the objects (i.e., **1110**$_1$, **1110**$_2$, **1115**, and **1120**) within the three-dimensional scene **1105**.

The first car object **1110**$_1$ and the second car object **1110**$_2$ may be moving within the three-dimensional scene **1105** while the circle object **1115** and the triangle object **1120** may be stationary within the three-dimensional scene **1105**. Consequently, in step **1005** of method **1000** the physics engine may classify the first car object **1110**$_1$ and the second car object **1110**$_2$ as dynamic objects by creating entries in a dynamic object queue for the first car object **1110**$_1$ and the second car object **1110**$_2$. Additionally, in step **1005** of method **1000**, the physics engine may classify the circle object **1115** and the triangle object **1120** as static objects by creating entries in a static object queue for the circle object **1115** and the triangle object **1120**.

FIG. **12** illustrates an exemplary dynamic object queue **1205** and an exemplary static object queue **1210**, according to embodiments of the invention. The dynamic object queue **1205** may be used to store information relating to dynamic objects within the three-dimensional scene, and the static object queue **1210** may be used to store information relating to static objects within the three-dimensional scene.

The dynamic object queue **1205** may have multiple entries or rows, each entry relating to a different dynamic data object within the three-dimensional scene **1105**. At step **1005** the physics engine may create entries in the dynamic data queue **1205** for each dynamic object within the three-dimensional scene **1105**. For example, as illustrated in FIG. **12**, the dynamic data queue **1205** may have a first entry **1215** which relates to the first car object **11110**$_1$, and a second entry **1220** which relates to the second car object **1110**$_2$. Furthermore, at step **1005** the physics engine may create entries in the static data queue **1210** for each static object within the three-dimensional scene. For example, as illustrated in FIG. **12**, the static data queue **1210** may have a first entry **1225** which relates to the circle object **1115** and a second entry **1230** which relates to the triangle object **1120**.

Next, at step **1010** of method **1000**, the physics engine may set a desirability factor and a complexity factor for each object within the three-dimensional scene. A desirability factor may be a constant which indicates, for example, how desirable it may be to classify an object as a dynamic object or to keep an object a dynamic object. The desirability factor may be determined or calculated based on how much the realism of the three-dimensional scene may be improved if an object is made dynamic or, conversely, how much realism may be lost if an object is made static. The desirability factor may also be set based on an object's proximity to other dynamic objects within the three-dimensional scene. If a dynamic object collides with another dynamic object, realism may be improved if both objects are classified as dynamic and consequently may move within the scene after the collision. Therefore, an object which is proximate to dynamic objects may have a higher desirability factor, while an object which is far from dynamic objects may have a lower desirability factor.

A complexity factor may indicate the complexity of physics calculations required to simulate physical phenomenon with respect to an object. For example, an object which may require relatively complex physics calculations to simulate physical phenomenon may have a relatively high complexity factor, while an object which may require less complex physics calculations may have a lower complexity factor.

FIG. **12** illustrates desirability factors and complexity factors for each of the objects within the three-dimensional scene **1105**. The first car object and the second car object may have high desirability factors (e.g., 8), because a car object may be commonly considered to be a moving or dynamic object and, therefore a realistic physics engine should have dynamic car objects. In contrast, the circle object and the triangle objects may have lower desirability factors in comparison to the first car object **1110**$_1$ and the second car object **1110**$_2$, because they may be considered inanimate objects or static objects.

However, the circle object **1115** may have a higher desirability factor (e.g., 4) than the triangle object **1120** (e.g., 3), because of the proximity of the circle object **1115** to dynamic objects (i.e., the first car object $\mathbf{1110}_1$ and the second car object $\mathbf{1110}_2$) and the increased likelihood of a collision between the car objects and the circle object **1115**.

FIG. **12** also illustrates complexity factors for each of the objects within the three-dimensional scene **1105**. The first car object $\mathbf{1110}_1$ and the second car object $\mathbf{1110}_2$ may have a higher complexity factors (e.g., 7) when compared to the complexity factors of the circle object **1115** and the triangle object **1120** (i.e., 3). The complexity factors for the car objects may be higher than the complexity factors for the triangle object **1120** and the circle object **1115**, because physics calculations may be more difficult for complex shapes such as a car than for simple shapes such as a triangle or a circle.

After setting the desirability factor and complexity factor for each object, the physics engine may proceed to step **1015** to flag essential dynamic objects. An essential dynamic object may be, for example, a dynamic object which is the main focus of the physical simulation or interacts with any objects considered to be the main focus of the physical simulation. According to embodiments of the invention, a game system may be restricted from changing a dynamic object which has been flagged as essential from a dynamic object to a static object.

For example, in a game system which simulates driving within a three-dimensional scene, the user of the game system may be controlling the movements of the first car object $\mathbf{1110}_1$. Consequently, for the game system to simulate realistic physical phenomenon, it may be essential that the game system move the first car object $\mathbf{1110}_1$ and perform certain physics operations (e.g., collision detection) with the first car object $\mathbf{1110}_1$. Therefore, at step **1015**, the physics engine may flag the first car object $\mathbf{1110}_1$ as an essential dynamic object. Furthermore, any objects connected to the essential object (e.g., the wheels of the first car object $\mathbf{1110}_1$) may also be flagged as essential objects.

FIG. **12** also illustrates an exemplary essential flag for dynamic objects within the three-dimensional scene **1105**. As stated above, the physics engine may flag the first car object $\mathbf{1110}_1$ as an essential dynamic object. This may be accomplished by setting a bit, for example, in the first entry **1215** of the dynamic object queue **1205** which corresponds to the first car object $\mathbf{1110}_1$. Dynamic objects which are not considered essential may not have their essential object flag bit set. For example, the second entry **1220** of the dynamic object queue **1205** which corresponds to the second car object $\mathbf{1110}_2$ does not have its essential object flag bit set.

After flagging the essential dynamic objects within the three-dimensional scene, the physics engine may proceed to step **1020** to perform physics operations for a predetermined period of time. The predetermined period of time may be, for example, the frame period (e.g., $1/30^{th}$-$1/60^{th}$ of a second). The physics operations may include all collision detection tests and collision handling operations related to the objects within the three-dimensional scene.

Next, at step **1025**, the physics engine may execute a performance analysis. The performance analysis may determine the workload presented to each processing element in the game system. The workload may be assessed, for example, over the period of time that the physics engine was performing physics operations for a single frame. However, other embodiments of the invention may asses the workload over a different period of time. Furthermore, the workload may be determined, for example, by examining the workload of processing elements within the multiple core processing element **100**, or by examining the operations of processing elements within the multiple core processing element network **200**.

The physics engine may determine the workload presented to the various processing elements, for example, by monitoring a performance counter for each processing element or monitoring the amount of data which passes through inboxes associated with each processing element.

In any case, at step **1030**, the image processing system may determine whether or not to adjust the number of static and dynamic objects based on the performance analysis executed in step **1025**. By adjusting the number of static or dynamic objects, the physics engine may vary the workload required to perform physics operations. An increase in the number of dynamic objects may increase the overall workload required to perform physics operations, while an increase in the number static objects may decrease the overall workload required to perform physics simulation.

For example, at step **1025** the physics engine may determine (based on the performance analysis) that all processing elements experienced a high workload and no spare processing cycles available on other processing elements. Therefore, the physics engine may determine at step **1030** that an adjustment to the physics workload is not necessary and, consequently, an adjustment in the number of objects which are considered static or dynamic is not necessary. Therefore, the physics engine may return to step **1010** to prepare for another cycle or frame of physics simulation. By returning to step **1010** without adjusting the number of static or dynamic objects, the physics engine may maintain a similar physics workload.

In contrast, the performance analysis, at step **1025**, may have revealed that some processing elements experienced a relatively low workload while the physics engine was performing physics operations. Therefore, the other processing elements may have spare processing cycles or processing bandwidth available for physics based operations. Consequently, at step **1030** the physics engine may adjust or increase the physics workload and may proceed to step **1035**.

At step **1035**, the physics engine may increase the physics workload by increasing the number of objects which are considered dynamic. The number of the objects which are to be reclassified from static to dynamic may be determined based on the amount of spare bandwidth available and, therefore, may be based on the results of the performance analysis. Which static objects become dynamic objects may be based on many factors. For example, which object is to become dynamic may be made based on the desirability factor of the object, the complexity factor of the object, and/or the performance analysis performed in step **1025**. According to one embodiment of the invention, an object may be chosen to become a dynamic object first based on the object's complexity factor and second based on the object's desirability factor.

Therefore, in the exemplary game system described above, if the physics engine detected available processing cycles or bandwidth on other processing elements, at step **1035** the physics engine may increase the number of objects which are considered dynamic. The physics engine may choose from the objects within the static object queue **1210**. Consequently, the physics engine may choose to re-classify the circle object **1115** or the triangle object **1120** from a static object to a dynamic object. According to embodiments of the invention, the circle object **1115** may be selected by the physics engine to be re-classified from a static object to a dynamic object, because the processing bandwidth may be available for an object as complex as the circle object **1115** and the circle object has a higher desirability factor than the triangle object.

Consequently, the physics engine may move the circle object **1115** from the static object queue **1210** to the dynamic object queue **1205**.

For example, as illustrated in FIG. **13**, the physics engine may delete the entry in the static object queue **1210** which corresponds to the circle object **1115**, and may add a third entry **1305** to the dynamic object queue **1205** which corresponds to the circle object **1115**.

By increasing the number of objects classified as dynamic and decreasing the number of objects classified as static, the physics engine may increase the physics workload for the next frame. Consequently, the physics engine may use some of the spare processing cycles or bandwidth to perform physics operations.

In contrast to processing elements experiencing a low workload, the performance analysis may have revealed that all of the processing elements experienced a high workload while the physics engine was performing physics operations. Therefore, the other processing elements may require additional processing cycles or processing bandwidth to perform operations (e.g., ray-tracing image processing operations). Consequently, at step **1030**, the physics engine may determine to adjust or decrease the physics workload and may proceed to step **1035**.

At step **1035**, the physics engine may decrease the physics workload by decreasing the number of objects which are considered dynamic. By decreasing the physics workload the physics engine may increase the processing bandwidth available for other processes. The number of objects which are to be reclassified from dynamic objects to static objects may be based on the amount of processing bandwidth needed and, therefore, may be based on the performance analysis executed in step **1025**. Which objects are to be reclassified from dynamic to static may be determined based on many factors. For example, a decrease in the number of dynamic objects may be made based on the essential flag, desirability factor, the complexity factor, and/or the performance analysis. According to one embodiment of the invention, an object may be reclassified from a dynamic object to a static object based initially on whether or not the object's essential flag bit is set, then based on the object's complexity factor, and lastly based on the object's desirability factor.

For example, in the exemplary game system described above, if the physics engine determines that other processing elements needed processing bandwidth, at step **1035** the physics engine may decrease the number of objects which are considered dynamic. The physics engine may choose from the objects within the dynamic object queue **1205** illustrated in FIG. **12**. Consequently, the physics engine may choose to re-classify either the first car object $\mathbf{1110_1}$ or the second car object $\mathbf{1110_2}$ from a dynamic object to a static object. However, upon examination of the first entry **1215** of the dynamic object queue **1205**, the physics engine may determine that the essential object flag for the entry corresponding to the first car object $\mathbf{1110_1}$ is set. Consequently, the physics engine may not change the first car object $\mathbf{1110_1}$ from a dynamic object to a static object. However, as illustrated in FIG. **12**, the second entry **1220** corresponding to the second car object $\mathbf{1110_2}$ does not have its essential object flag set. Therefore, the physics engine may reclassify the second car object $\mathbf{1110_2}$ from a dynamic object to a static object. Consequently, as illustrated in FIG. **14**, the physics engine may delete the entry in the dynamic object queue **1205** which corresponds to the second car object $\mathbf{1110_2}$ and may add a third entry **1405** to the static object queue **1210**. The third entry **1405** in the static object queue **1210** may correspond to the second car object $\mathbf{1110_2}$.

By reducing the number of objects classified as dynamic and increasing the number of objects classified as static, the physics engine may reduce the physics workload for the next frame. Consequently, the physics engine may free processing cycles or bandwidth for other game system operations (e.g., ray-tracing image processing).

Although embodiments of the invention were described as changing an objects classification as static or dynamic by moving the object from a static object queue to a dynamic object queue or from a dynamic object queue to a static object queue, embodiments of the invention are envisioned that may not require separate object queues. For example, according to another embodiment of the invention, single queue may contain entries for every object within the three-dimensional scene. Furthermore, a bit within each entry may indicate if an object is a static object or a dynamic object.

CONCLUSION

By reclassifying objects from static objects to dynamic objects or from dynamic objects to static objects, a physics engine may control the amount of workload required to perform physics operations for a given frame. Furthermore, by periodically examining the workload presented to other processing elements, a physics engine may determine when to increase or decrease the physics workload.

While the foregoing is directed to embodiments of the present invention, other and further embodiments of the invention may be devised without departing from the basic scope thereof, and the scope thereof is determined by the claims that follow.

What is claimed is:

1. A method of varying physics workload by varying classification of a plurality objects within a three-dimensional scene, comprising:

classifying each of the plurality of objects within the three-dimensional scene as a respective first one of at least a static object and a dynamic object;

performing physics based operations on the objects for a timeframe;

analyzing, by operation of one or more computer processors, workload experienced by processing elements during the timeframe; and

based on the workload experienced by the processing elements during the timeframe, reclassifying at least one of the objects within the three-dimensional scene as a respective second one of at least the static object and the dynamic object.

2. The method of claim **1**, wherein reclassifying at least one of the objects comprises at least one of:

reclassifying an object from a dynamic object to a static object; and

reclassifying an object from a static object to a dynamic object.

3. The method of claim **1**, wherein at least one of the plurality of objects within the three-dimensional scene is classified as an essential dynamic object, wherein the static or dynamic classification of the essential dynamic object cannot be modified.

4. The method of claim **1**, further comprising: assigning a complexity factor to each of the objects within the three-dimensional scene, and wherein reclassifying at least one of the objects within the three-dimensional scene is based at least in part on the workload experienced by processing elements and the complexity factor assigned to the object.

5. The method of claim **4**, further comprising: assigning a desirability factor to each of the objects within the three-dimensional scene, and wherein reclassifying at least one of the objects within the three-dimensional scene is based at least in part on the workload experienced by processing elements, the complexity factor assigned to the object, and the desirability factor assigned to the object.

6. The method of claim **1**, wherein physics based operations comprise at least one of collision detection tests and collision handling operations in the three-dimensional scene, and wherein the workload experienced by the processing elements comprises image processing operations for the three-dimensional scene.

7. The method of claim **1**, wherein performance analysis comprises monitoring at least one of a performance counter for a processing element or an inbox associated with a processing element.

8. The method of claim **1**, wherein the timeframe is the period of a frame in an image processing system.

9. A computer readable medium containing a program which, when executed, performs an operation comprising:

classifying each of a plurality of objects within a three-dimensional scene as a respective first one of at least a static object and a dynamic object;

performing physics based operations on the objects for a timeframe;

analyzing workload experienced by processing elements during the timeframe; and

based on the workload experienced by the processing elements during the timeframe, reclassifying at least one of the objects within the three-dimensional scene as a respective second one of at least the static object and the dynamic object.

10. The computer readable medium of claim **9**, wherein reclassifying at least one of the objects comprises at least one of:

reclassifying an object from a dynamic object to a static object; and

reclassifying an object from a static object to a dynamic object.

11. The computer readable medium of claim **9**, wherein at least one of the plurality of objects within the three-dimensional scene is classified as an essential dynamic object, wherein the static or dynamic classification of the essential dynamic object cannot be modified.

12. The computer readable medium of claim **9**, wherein the operation further comprises further comprising:

assigning a complexity factor to each of the objects within the three-dimensional scene, and wherein reclassifying at least one of the objects within the three-dimensional scene is based at least in part on the workload presented to processing elements and the complexity factor assigned to the object.

13. The computer readable medium of claim **12**, wherein the operation further comprises: assigning a desirability factor to each of the objects within the three-dimensional scene, and wherein reclassifying at least one of the objects within the three-dimensional scene is based at least in part on the workload presented to processing elements, the complexity factor assigned to the object, and the desirability factor assigned to the object.

14. The computer readable medium of claim **9**, wherein physics based operations comprise at least one of collision detection tests and collision handling operations in the three-dimensional scene, and wherein the workload experienced by the processing elements comprises image processing operations for the three-dimensional scene.

15. A system, comprising:

a plurality of objects within a three-dimensional scene;

a first processing element; and

a second processing element configured to classify each of the plurality of objects as a respective first one of at least a static object and a dynamic object, perform physics based operations on the objects for a timeframe, analyze workload experienced by the first processing element during the timeframe, and, based on the workload experienced by the first processing element during the timeframe, reclassifying at least one of the objects within the three-dimensional scene as a respective second one of at least the static object and the dynamic object.

16. The system of claim **15**, further comprising:

a dynamic object queue; and

a static object queue; and

wherein the second processing element classifies the plurality of objects as one of at least the static object and the dynamic object by storing the object in at least one of the dynamic object queue or the static object queue.

17. The system of claim **16**, wherein the second processing element reclassifies at least one of the objects by doing at least one of:

moving an object from the dynamic object queue to the static object queue; and

moving an object from the static object queue to the dynamic object queue.

18. The system of claim **15**, wherein

the system is a game system; and

wherein the first processing element is configured to perform image processing operations.

19. The system of claim **15**, wherein the second processing element analyzes the performance of the first processing element by monitoring at least one of a performance counter for the first processing element or an inbox associated with the first processing element.

20. The system of claim **15**,

wherein the first processing element comprises a plurality of threads;

wherein the threads of the first processing element are configured to perform image processing operations for the timeframe;

wherein the second processing element comprises a plurality of threads;

wherein the threads of the second processing element are configured to perform physics operations for the timeframe; and

wherein reclassifying at least one of the objects within the three-dimensional scene results in at least one of a portion of the plurality of threads of the first processing element performing physics operations for a subsequent timeframe or a portion of the plurality of threads of the second processing element performing image processing operations for a subsequent timeframe.

* * * * *